TURING

Cracking the Coding Interview

150 Programming Questions and Solutions

5th edition

程序员面试金典

第5版

[美] Gayle Laakmann McDowell 著　李琳骁 漆犇 译

全方位揭示**微软、苹果、谷歌**等IT名企招聘秘密

亚马逊超级畅销书，雄踞排行榜数年之久

数十万程序员求职成功的敲门砖



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



Gayle Laakmann McDowell

美国求职咨询网站CareerCup.com创始人兼CEO，是一位著名软件工程师，曾在微软、苹果与谷歌任职。早先，她自己就是一位十分成功的求职者，通过了微软、谷歌、亚马逊、苹果、IBM、高盛等多家最著名企业极其严苛的面试过程。工作以后，她又成为一位出色的面试官。在谷歌任职期间，她还是该公司资深面试官及招聘委员会成员，积累了相当丰富的面试经验。除此书外，她还著有《金领简历：敲开苹果、微软、谷歌的大门》。

李琳骁

主要从事嵌入式Linux内核/驱动开发，并关注IT、开放源码和安防监控等领域。业余时间以技术翻译为乐，时而客串编辑，好为爱书挑错，渴求完美，却也常因“小”失大，不得读书要领。翻译或参与翻译了《Linux命令详解手册》《编程人生》《编程大师访谈录》等图书。网络ID为leal，管理Vim、Android等豆瓣小组，个人站点：http://linxiao.net。

漆犇

毕业于中国地质大学，拥有十余年软件开发、测试及流程管理经验，热衷翻译，已出版译作包括《Linux/Unix设计思想》《金领简历：敲开苹果、微软、谷歌的大门》等书。目前定居于美国西雅图，在微软Windows Phone开发中心从事与WP应用开发者相关的项目管理事务。

TURING

Cracking the Coding Interview

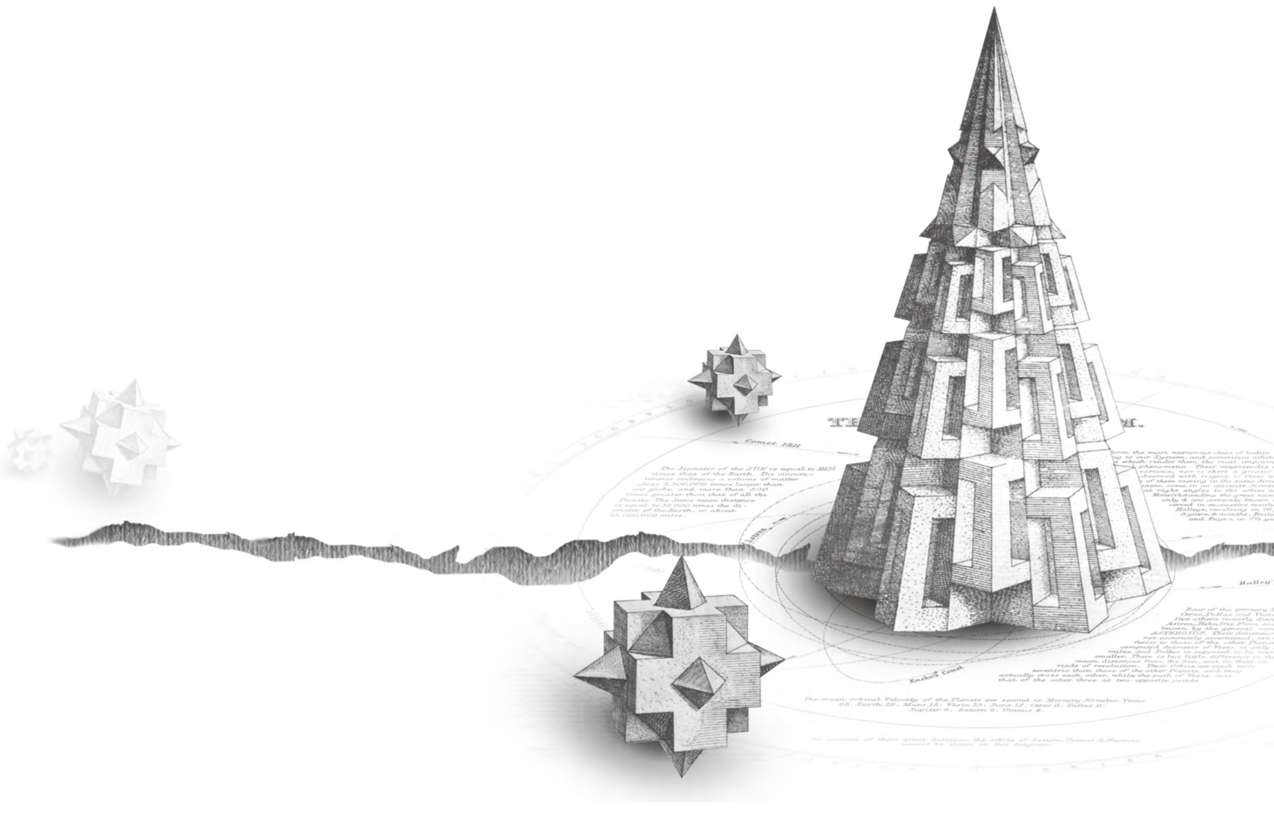
150 Programming Questions and Solutions

5th edition

程序员面试金典

第 5 版

[美] Gayle Laakmann McDowell 著　　李琳骁 漆犇　译



人民邮电出版社
北　京

图书在版编目（CIP）数据

程序员面试金典 : 第5版 / (美) 麦克道尔
(McDowell,G.L.) 著 ; 李琳骁, 漆犇译. -- 北京 : 人
民邮电出版社, 2013.11
书名原文: Cracking the coding interview:150
programming questions and solutions,fifth edition
ISBN 978-7-115-33291-2

Ⅰ. ①程… Ⅱ. ①麦… ②李… ③漆… Ⅲ. ①程序设
计－工程技术人员－资格考试－自学参考资料 Ⅳ.
①TP311.1

中国版本图书馆CIP数据核字(2013)第245017号

内 容 提 要

本书是原谷歌资深面试官的经验之作，层层紧扣程序员面试的每一个环节，全面而详尽地介绍了程序员应当如何应对面试，才能在面试中脱颖而出。第 1～7 章主要涉及面试流程解析、面试官的幕后决策及可能提出的问题、面试前的准备工作、对面试结果的处理等内容；第 8～9 章从数据结构、概念与算法、知识类问题和附加面试题 4 个方面，为读者呈现了出自微软、苹果、谷歌等多家知名公司的 150 道编程面试题，并针对每一道面试题目，分别给出了详细的解决方案。

本书适合程序开发和设计人员阅读。

◆ 著　　　　[美] Gayle Laakmann McDowell
译　　　　李琳骁　漆　犇
责任编辑　丁晓昀
执行编辑　李　鑫　陈婷婷
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：23.25
字数：549千字　　　　2013年 11月第 1 版
印数：1 – 5 000册　　　2013年 11月北京第 1 次印刷
著作权合同登记号　图字：01-2012-1951号

定价：59.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

序

亲爱的读者：

我先做个自我介绍。

我不是什么招聘人员。我只是一名软件工程师。正因如此，我深知大家要在面试现场迅速想出精妙算法并在白板上写下完美代码的感受。之所以能感同身受，是因为我与你们有过同样的经历，我参加过谷歌、微软、苹果、亚马逊以及其他诸多公司的面试。

而且，我也当过面试官，让求职者做同样的事情。我还筛选过成千上万份简历，在其中“上下求索”，希望挑出那些或许能在面试难关中脱颖而出的工程师。在谷歌时，我与招聘委员会的同僚有过激烈争辩，探讨某位求职者是否达到了录用要求。我对招聘各环节了如指掌，相关经验也很丰富。

而现在，我亲爱的读者，你也许要在明天、下周或是明年去迎接面试挑战。你可能已经拿到或者正在攻读计算机科学或相关专业的学位。本书并不打算给大家重温有关二叉查找树的基本知识，或者该如何遍历链表。想必你已经掌握这些内容；倘若没有，还请先找些数据结构的基础资料仔细研读。

本书旨在帮助你加深对计算机科学基础知识的理解，并学会该如何运用这些基础知识，成功闯过技术面试这一关。

本书在第四版的基础上做了大量更新，增补篇幅达 200 多页。第五版添补了不少面试题，修订了部分原有题目的解法，并新增了几个章节和其他内容。欢迎访问我们的网站，你可以跟其他求职者互通有无，发现新天地。

与此同时，我也感到无比兴奋，你一定能从本书中学到新的技能。充分的准备会让你在技术和人际沟通技能等诸多方面更进一步。不管最终结果如何，只要拼尽全力，无怨无悔！

请各位读者务必用心研读本书前面的介绍性章节，其中的要点和启示也许可以决定你的面试结果，“录用”与“拒绝”就在一线之间。

此外，切记——面试非易事！根据我在谷歌多年面试的经历，我留意到有些面试官会问一些"简单的"问题，有些则会专挑难题来问。但是你知道吗？面试中碰到简单的问题，也不见得就能轻松过关。完美解决问题（只有极少数求职者才能做到！）不是公司录用你的关键，只有题答得比其他求职者更出色才能让你脱颖而出。所以，碰到棘手的难题也不要惊慌，或许其他人一样觉得很难。

请努力学习，不断实践。祝你好运！

盖尔·拉克曼·麦克道尔
CareerCup.com 创始人兼 CEO
《金领简历：敲开苹果、微软、谷歌的大门》及本书作者

致　　谢

生命中很多事情都离不开团队合作，这本书也不例外。在创作本书的过程中，我得到了很多人的帮助，尽管涌泉都难以回报，我还是想在此聊表寸心。

首先，我要感谢我的丈夫约翰，他是我最坚强的后盾，让我有勇气将此书一改再改并接连修订了五版。如果没有他的支持，我很可能就做不到这一点。

其次，我要感谢家母，她让我认识到编程无比重要，而写出优美流畅的文字更为重要。毫无疑问，她是一位无与伦比的工程师、企业家，最重要的是，她是一位伟大的母亲。

接下来我要感谢诸多好友的鼓励，尤其是加尔顿·英格里绪。不管是我需要帮助还是听我发牢骚，她总是默默陪在我身边，一如既往地支持我。

最后，我还要感谢那些给予回复与建议的读者：谢谢你们！我要特别感谢维尼特·萨哈和普拉雷·瓦玛，他们细致入微地审阅了书中的每一道题，用心之极，佩服不已。相信你们的同事和主管一定会为有这么出色的伙伴而骄傲。

再次深表谢意！

前　　言

招聘中的问题

讨论完招聘事宜，我们又一次沮丧地走出会议室。那天，我们重新审查了十位“过关”的求职者，但是全都不堪录用。我们很纳闷，是我们太过苛刻了吗？

我尤为失望的是，我推荐的一名求职者也被拒了。他是我以前的学生，以高达 3.73 的平均分（GPA）毕业于华盛顿大学，这可是世界上最棒的计算机专业院校之一。此外，他还完成了大量的开源项目工作。他精力充沛、富于创新、踏实能干、头脑敏锐，不论从哪方面来看，他都堪称真正的极客。

但是，我不得不同意其他招聘人员的看法：他还是不够格。就算我的强力推荐可以让他侥幸过关，在后续的招聘环节还是会失利，因为他的硬伤太多了。

尽管面试官都认为他很聪明，但他答题总是磕磕绊绊的。大多数成功的求职者都能轻松搞定第一道题（这一题广为人知，我们只是略作调整而已），可他却没能想出合适的算法。虽然他后来给出了一种解法，但没有提出针对其他情形进行优化的解法。最后，开始写代码时，他草草地采用了最初的思路，可这个解法漏洞百出，最终还是没能搞定。他算不上表现最差的求职者，但与我们的“录用底线”却相去甚远，结果只能是铩羽而归。

几个星期后，他给我打电话询问反馈意见，我很纠结，不知该怎么跟他说。他需要变得更聪明些吗？不，他其实智力超群。做个更好的程序员？不，他的编程技能和我见过的一些最出色的程序员不相上下。

跟许多积极上进的求职者一样，他准备得非常充分。他研读过 Brian W. Kernighan 和 Dennis M. Ritchie 合著的《C 程序设计语言》，麻省理工学院出版的《算法导论》等经典著作。他可以细数很多平衡树的方法，也能用 C 语言写出各种花哨的程序。

我不得不遗憾地告诉他：光是看这些书还远远不够。这些经典学院派著作教会了人们错综复杂的研究理论，对程序员的面试却助益不多。为什么呢？容我稍稍提醒你一下：即使从学生时代起，你的面试官们其实都没怎么接触过所谓的红黑树（Red-Black Trees）算法。

要顺利通过面试，就得“真枪实弹”地做准备。你必须演练真正的面试题，并掌握它们的解题模式。

这本书就是我根据自己在顶尖公司积累的第一手面试经验提炼而成的精华。我曾经与数百名

求职者有过“交锋”，本书可以说是我面试几百位求职者的结晶。同时，我还从成千上万求职者与面试官提供的问题中精挑细选了一部分。这些面试题出自许多知名的高科技公司。可以说，这本书囊括了 150 道世界上最好的程序员面试题，都是从数以千计的好问题中挑选出来的。

我的写作方法

本书重点关注算法、编码和设计问题。为什么呢？尽管面试中也会有“行为问题”，但是答案会随个人的经历而千变万化。同样，尽管许多公司也会考问细节（例如，“什么是虚函数？”），但通过演练这些问题而取得的经验非常有限，更多地是涉及非常具体的知识点。本书只会述及其中一些问题，以便你了解它们“长”什么样。当然，对于那些可以拓展技术技能的问题，我会给出更详细的解释。

我的教学热情

我特别热爱教学。我喜欢帮助人们理解新概念，并提供一些学习工具，从而充分激发他们的学习热情。

我第一次“正式”的教学经验是在美国宾夕法尼亚大学就读期间，那时我才大二，担任本科计算机科学课程的助教（TA）。我后来还在其他一些课程中担任过助教，最终在大学里推出了自己的计算机科学课程，也就是给大家教授一些实际的“动手”技能。

在谷歌担任工程师时，培训和指导“Nooglers”（意指谷歌新员工。没错，他们就是这么称呼新人的！）是我最喜欢的工作之一。后来，我还利用“20%自由支配时间”在华盛顿大学教授计算机科学课程。

《程序员面试金典》、《金领简历》和 CareerCup.com 网站都能充分体现我的教学热情。即便是现在，你也会发现我经常出现在 CareerCup.com 上为用户答疑解惑。

请加入我们的行列吧！

Gayle Laakmann McDowell

目　　录

第 1 章 面试流程

- 概述
- 面试题的来源
- 准备时间表与注意事项
- 面试评估流程
- 答题情况
- 着装规范
- 十大常见错误
- 常见问题解答

1.1 概述

大多数公司的面试流程其实都大同小异。本章会简述面试流程，以及企业到底想招募什么样的人才。这些信息将指导你如何做好面试准备，以及在面试过程中和面试结束后该如何应对。

收到面试通知后，你通常得先经历一次**筛选面试**（screening interview），一般通过电话进行。顶尖高校的应届毕业生则可能需要参加现场的筛选面试。

不要因“筛选面试”这个词儿而掉以轻心，筛选面试也很有可能涉及编码与算法问题，要求不见得比现场面试低。如果不确定它是不是技术筛选面试，不妨问问招聘助理面试官是什么来头，若是工程师，那十有八九会与技术相关。

许多公司会在面试中运用在线同步文档编辑系统，但也有可能让你直接在纸上写好代码，然后在电话里念给他们听。有些面试官甚至还会给你留“家庭作业”，或是要求你用电子邮件将写好的代码发给他们。

在**现场面试**（on-site interview）之前，通常会有一两轮筛选面试。现场面试大概有4到6轮，其中一轮可能是午餐面试。当然，午餐面试比较随意，面试官一般不会问你技术问题，甚至不会纳入面试评价范畴。但同时，这也是难得的好机会，你可以跟面试官探讨自己感兴趣的问题，了解公司的企业文化。其他几轮面试主要涉及技术问题，包括编码和算法等。此外，你可能还要回答与简历相关的问题。

面试结束后，面试官们会聚在一起讨论你的表现，或者提交书面评价。大多数情况下，公司招聘人员都会在一周内给你回复，告知应聘进展。

要是已经望穿秋水等了一个多星期，你也可以主动询问进展。就算招聘人员没有回应，也并不表示你被拒了（至少大的高科技公司是这样，其实几乎所有公司都是如此）。我再重复一次：没有回应表示你的应聘结果还是未知数。当然，人们都希望招聘方在得出最终结论时，及时通知求职者。

拖拖拉拉的情况确实有。等不及的话，不妨问问相关招聘人员，但务请有礼有节。招聘人员和我们一样，他们很忙，有些人会因此容易忘事。

1.2　面试题的来源

求职者经常会问我，某些公司最近都喜欢问哪些面试题？他们总以为面试题会应时而变。实际上，公司本身对面试题并没有什么倾向，这完全取决于面试官的个人喜好。容我解释一下。

在大公司里，面试官通常需要先参加一些面试培训课程。在谷歌，担任面试官之前，我先参加了一次由外部公司提供的专门培训。培训课程为期一天，有一半时间侧重于法律层面的事务，比如，面试官不能探问求职者的婚姻状况，不得询问种族，等等。另一半时间则在探讨如何应对"刺头"求职者，比如当问及编码问题或其他令求职者认为是在"羞辱"自己的问题时，要是求职者"暴跳如雷"，该怎么应对。培训过后，我又实地观摩了两次真正的面试，然后就开始独自面试了。

就是这样。我们受过的培训也不过如此，其实所有公司都大同小异。

根本就不存在什么"谷歌官方面试题清单"，也从来没有人要求我一定要问哪些特定的问题，或者必须避开哪些话题。

那我的面试题从何而来呢？其实，来源和大家一样。

面试官也当过求职者，他们会借用自己当年被拷问过的题目。又或者，有些面试官也会彼此交换题库。还有些人喜欢上网找问题，比如CareerCup.com网站。有些面试官也可能从上述渠道收集面试题，并或多或少做些调整。

就算真有公司给面试官准备好问题清单，这种情况也并不多见。面试官通常也会自行挑选问题，而且大家往往会有五六个常用的备选题目。

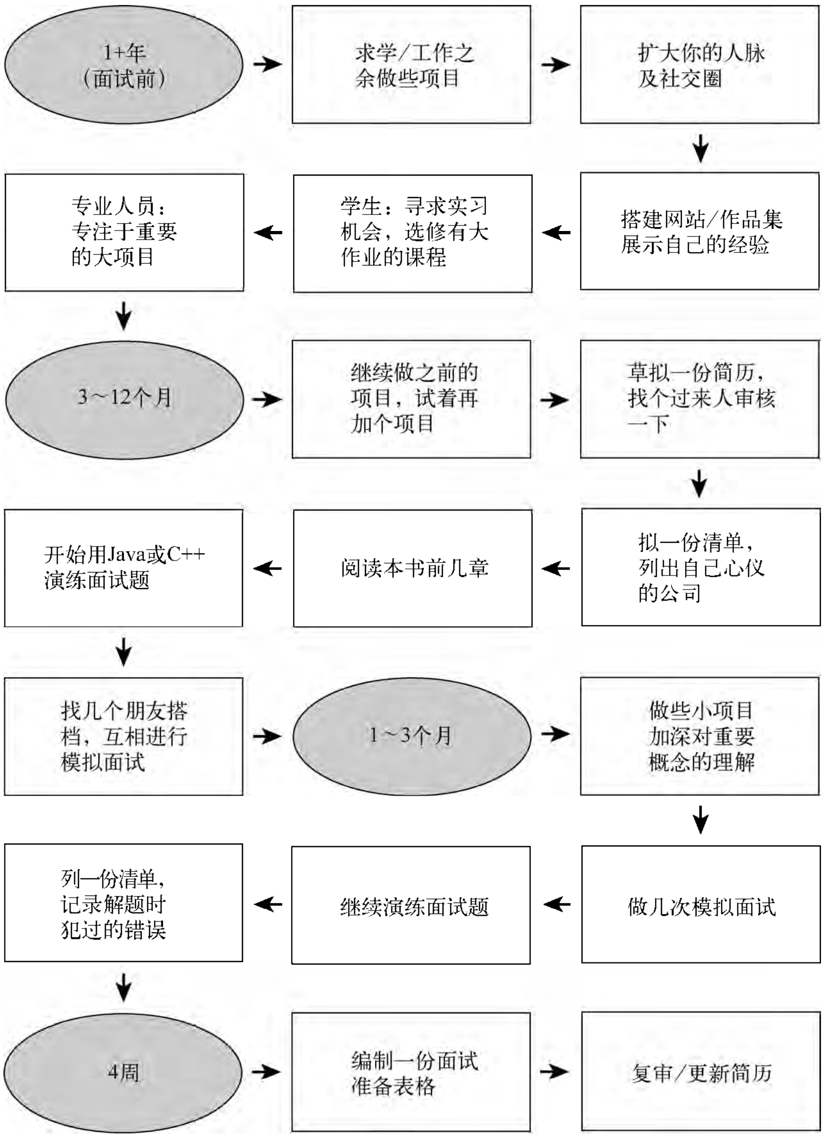
因此，下次在你想知道谷歌"最近"都问些什么问题的时候，不妨先停下来想一想。谷歌与亚马逊的面试题其实没什么不同，他们需要的都是软件开发人才。至于面试题是不是"最近流行的"也就更无关紧要了。万变不离其宗，因为这本来就得靠面试官自己去把握。

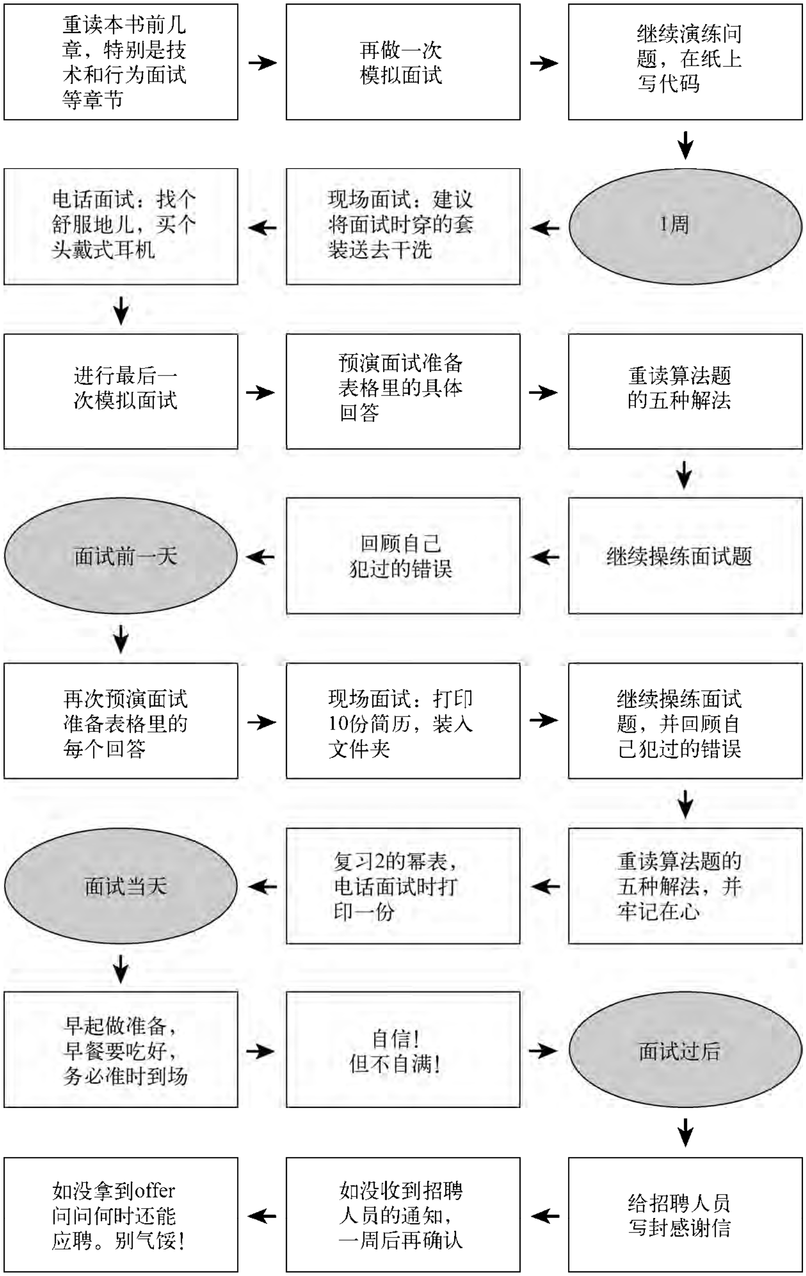
当然，总体上，不同的公司在风格上存在差异。互联网公司往往会提些系统设计方面的问题，而那些使用数据库的公司则明显偏爱数据库方面的问题。然而，大部分面试题无外乎就是数据结构和算法之类的，任何公司都会问到。

1.3　准备时间表与注意事项

“台上一分钟，台下十年功”，事实上你的面试表现取决于你的功底——离不开多年的积淀。你需要刚好具备能为公司所用的技术经验，然后还要准备好在面试中解决实际的技术问题。下面的时间表和流程图可以给你一些启发。

如果你起步比较晚，也不用担心。“尽人事，知天命”，请安心准备，祝你好运！





1.4　面试评估流程

招聘人员可能会告诉你，他们主要考查四个方面：工作经验、企业文化契合度、编程技能及

分析能力。这四个方面相辅相成，但在决定录用与否时，分量最重的通常还是编程技能和分析能力（或者看你是否聪明）。这也是为什么本书的主要篇幅都在探讨如何提升编程与算法技能。

当然，虽说编程与算法技能往往最为重要，但并不表示你可以忽视其他两个方面。

一旦进入大型科技公司的面试环节，你之前的工作经验就不是特别重要了，但它可能会左右面试官对你的看法。比如，如果你说起以前写的某个复杂程序的精彩之处，面试官很有可能会想："哇，她可真聪明！"一旦他认定你智力超群，可能就会下意识地忽略你所犯的小错误。总之，面试并不会十分精确，对某些"软问题"做好充分准备会大有裨益。

创业公司比大公司更看重企业文化契合度（或你的个性，主要看是否与公司合拍）。举个例子，如果公司的企业文化鼓励员工独立做决定，那么喜欢听从指导的人就不太适合了。

此外，求职者因为过于自大、巧辩或抵触而被淘汰的情况也并不少见。我就遇到过，有位求职者对我提问的用词吹毛求疵，并抱怨这导致他解题不太顺利，后来他还对我的引导方式心生不满。这种"抵触心太重"的表现其实也是一个警示，果然，其他面试官对他的感觉也很不好。最后他被淘汰了。谁会愿意跟这种人一起共事呢？

所以，你应该注意以下几点。

- 如果人们都认为你骄傲自大、过于狡辩，或有其他负面评价，那你最好在面试中收敛一下。个性不讨喜的话，哪怕你的表现再好，也可能会被拒。
- 准备一些与简历相关的问题。虽然这不是最重要的因素，但也不能掉以轻心。稍微花点时间准备就能起到很好的效果，做到"四两拨千斤"。
- 把主要精力用在编程与算法问题上。

最后，我还是要再强调一遍，面试并不会十分精确。你的表现可能会有失水准，招聘委员会（或不管是谁）有时候也会做出错误判断。就像任何群体一样，招聘委员会也可能会被某位主导人物的观点所左右。这也许不公平，但这就是生活。

记住——这次被拒绝并不代表永远。一年内你还可以重新应聘，很多求职者都有过失利后再成功的经历。

不要气馁，失败是成功之母。

1.5　答题情况

有则谣传流传甚广且颇具迷惑性：求职者必须答对全部问题才会被录用。事实绝非如此。

首先，面试题的答案很难用"正确"和"错误"去简单评判。我个人在评估求职者的面试表现时，一般不会只看他们答对了几道题。相反，我会考量其最终解法是否最优，用时多久，代码整洁与否。这不只是单纯的是非判断，还要综合考虑很多因素。

其次，你的面试表现还会拿来跟其他求职者作比较。比如说，你用15分钟出色地解决了一道题，而另一个人不到5分钟就搞定了一道比较容易的题，是否就意味着那个人的表现比你好呢？也许是，但也未必。很自然，面试官出的题越简单，他们越是希望你尽快给出最佳答案。但要是题目很难，他们也不会指望你能答得又快又好，毕竟，出点纰漏也是在所难免的。

我在谷歌评估过数千名求职者的面试资料，其中只有一位求职者的面试表现堪称“完美无瑕”。其他人，包括最后被录用的几百个幸运儿，都或多或少犯过一些错。

1.6　着装规范

软件工程师一般都穿得比较随意。这一点从面试的着装规范也看得出来。参加面试时，推荐做法是穿得比同级别员工稍好一点。

以下是我给软件工程师（及测试人员）的面试着装建议，意在让大家找到一个“平衡点”：不要穿得过于正式，也不要太随意。其实，有很多人还是穿着牛仔裤和T恤衫参加创业公司或大公司的面试，也不会有什么问题。毕竟，公司不是看你穿什么，而是看你的编程水平。

	创业公司	微软、谷歌、亚马逊、Facebook等科技巨头	非科技公司（包括银行）
男　性	卡其裤、休闲裤或整洁得体的牛仔裤。Polo衫或礼服衬衫	卡其裤、休闲裤或整洁得体的牛仔裤。Polo衫或礼服衬衫	套装，不打领带（可带一条领带以防万一）
女　性	卡其裤、休闲裤或整洁得体的牛仔裤。大方得体的上衣或毛衣	卡其裤、休闲裤或整洁得体的牛仔裤。大方得体的上衣或毛衣	套装，或得体的休闲裤配整洁的上衣

这些只是指导建议，具体还要参考公司的企业文化。此外，如果你应聘的是项目经理、开发主管或其他管理层职位，面试时最好还是穿得正式一点。

1.7　十大常见错误

错误一：只在计算机上练习

如果你正准备参加海洋游泳比赛，你会只在泳池里练习吗？应该不会。你得去体验大风大浪及海洋里各种情况带来的影响。所以，你肯定会希望到海洋中实地训练。

在计算机上借助编译器演练面试题就像只在泳池里练习一样。抛开这个环境吧，让我们拿出纸和笔。你可以在写好全部代码并做过人工测试之后，再在计算机上用编译器进行验证。

错误二：不做行为面试题演练

很多求职者将全部时间花在演练技术问题上，而忽视了行为面试题。你猜怎么着？面试官可是两者都会考查的。

而且不止于此，你回答行为问题的表现其实还会左右面试官对你技术能力的看法。行为问题的准备工作其实相对比较轻松，而且容易达到事半功倍的效果。用心回顾你以往的项目和经历，然后准备一些小故事。

错误三：不做模拟面试训练

假设你要准备一场重大演讲，所有同事和相关人员都将列席，而且它还关乎你的未来。要是只在头脑里无声地练习演讲，到了真正演讲时，你肯定会发狂的。

光是纸上谈兵，不做模拟面试也会陷入同样的境地。如果你是一名工程师，肯定认识不少同行。不妨找个朋友帮你做模拟面试。作为回报，你也可以给他当一回面试官。

错误四：试图死记硬背答案

死记硬背答案最多只能解决一些特定问题，但是一碰到新的题，你可能就傻眼了。而且，基本上你不太可能碰上出自本书的题目。

最靠谱的做法就是，不看答案，先把书里的题全部认真做一遍。这样你才有可能练就各种技能和技巧，从容应对新问题。就算最后你只能大概复习一下为数不多的题，这种做法也会对你很有帮助。质量胜于数量。

错误五：不大声说出你的解题思路

透露个秘密：面试官才不会知道你心里想什么。因此，面试时默不作声，我根本无法了解你的思路。假如你沉默时间过长，我还会误以为你毫无进展。你得多出声，没准说着说着就找到了解法。请大声说出解题的思路，这样面试官就会知道你还在处理这个问题，没有卡壳。

这么做还有个好处就是不至于跑题，从而有助于你尽快找到解法。当然，最大的作用就是突显你强大的沟通能力。何乐而不为呢？

错误六：过于仓促

写程序不是什么竞赛，面试也不是，所以解题时不要太过仓促。代码写得太草率容易出问题，也说明你这个人不够细心。请放慢节奏，有条不紊，多做测试，问题考虑得周全些。这么一来，最终你反而能更高效地给出答案，错误也会少一些。

错误七：代码不够严谨

其实每个人都写得出完美的代码，但有时我们还是会在面试中写出错误百出的程序，不是吗？代码冗余、数据结构乱七八糟（比如，缺少面向对象设计）等等，这些都是常见错误！写代码时，不妨设想一下你是在处理实际问题，要注重可维护性。将代码划分成不同的子程序，并精心设计数据结构来处理相应的数据。

错误八：不做测试

在日常工作中，你不可能不做任何测试就提交代码，既然如此，为什么要在面试中省略这一步呢？写完代码后，请“运行”（或者审查）一下程序来验证结果。或者，在处理复杂问题时，你还可以边写代码边测试。

错误九：修正错误漫不经心

程序总会有bug，这就是生活或编程的本来面目。只要用心测试你的代码，bug也许就会现出原形。那也不错。

不过，重要的是发现bug时，你必须三思而后行，修正之前先确定出错原因。有些求职者看到传入特定参数时函数返回`false`而不是`true`，会直接将返回值取反，接着检查问题是否得到修正。当然，偶尔他们也能瞎猫碰上死耗子，但实际上如此仓促行事往往会导致更多的bug，同时也反映出你这个人比较粗心大意。

有bug其实很正常，但胡乱修改代码却很严重。

错误十：轻言放弃

我知道面试题都很难，但不难怎么显出求职者的水平呢。你会迎难而上还是轻言放弃？态度很重要，面试官都喜欢那些不畏挑战、迎难而上解决问题的求职者。毕竟，面试本来就不简单。

所以，碰到棘手的问题请不要惊慌，也不要轻言放弃。

1.8　常见问题解答

1. 碰到熟悉的问题时应该如实相告吗？

是的！碰到熟悉的问题，当然要告诉面试官！有些人会觉得这很傻——要是熟悉这个问题（并知道答案），岂不是如虎添翼，对吧？其实，未必如此。

我们力荐你如实相告的理由如下。

(1) 彰显你的诚实品质。这能反映出你的诚信——可以大大加分！要知道面试官可是在默默地考察你，看你够不够格成为他未来的同事。我不知道你个人怎么想，反正我是喜欢和实在人一起共事。

(2) 这个问题可能略有改动。你不会想冒这个险给个错误答案吧？

(3) 如果你将正确答案脱口而出，面试官会觉得很可疑。面试官当然知道题目的难度。但如果你佯装磕磕绊绊地答题，则很有可能夸张过度，而显得你这个人很不诚实。

2. 该使用哪种编程语言？

很多人都会建议说用自己最得心应手的语言，其实理想情况下，你应该使用面试官最熟悉的语言。我一般会推荐使用C、C++或Java，因为大多数面试官都熟悉这三种语言。我个人偏好Java（除非涉及C/C++问题），因为用Java编写程序效率比较高，而且写出来的程序简单易懂，哪怕平时用惯C++的人看Java程序也不会有太大难度。有鉴于此，本书基本上都用Java来解题。

3. 面试结束后我没有收到回复，是被拒了吗？

不是的。真要被拒的话，公司一般都会给你通知。面试结束后短时间内没有收到回复并不代表什么。你可能表现得很不错，但招聘人员不巧度假去了，没能及时处理。公司可能正在进行部门重组，具体该招多少人尚无定论。又或者，你确实表现得不怎么样，但碰巧遇到了一个办事拖拉或者特别忙的招聘人员，他没能及时答复你。当然，也会有一些奇怪的公司。“嗯，既然我们不打算录用这个求职者，那就没必要给他回复。”所以，一切取决于公司本身。但你可以发邮件或打电话跟踪后续进展。

4. 被拒之后我还能重新申请吗？

当然可以了，不过通常需要等上一段时间（半年至一年）。上一次的糟糕表现一般不会影响下一次面试。很多人都被微软、谷歌拒过，但他们后来还是顺利过关了。

第 2 章

面试揭秘

- 微软面试
- 亚马逊面试
- 谷歌面试
- 苹果面试
- Facebook面试
- 雅虎面试

对于多数求职者而言，面试好似一个迷局。你去了，见了几个面试官，答了一堆问题，然后，或两手空空离开，或幸运地拿到录用通知。

你有没有想过：

- 面试结果是怎么得出的？
- 面试官会不会互相交流？
- 公司最看重哪些方面？

好了，不用再挖空心思、再三思索了，我来告诉你。

在本章，我们邀请了来自顶尖科技公司（微软、亚马逊、谷歌、苹果、Facebook及雅虎）的面试专家来为大家答疑解惑，揭秘面试中的那些事儿。

这些专家会让我们了解各家公司的面试流程，帮助还原那些发生在面试会议室之外的事情，以及面试结束后的事项。

这些专家还会告诉我们各家公司面试流程的不同之处。比如，亚马逊的“调杆员”[①]是怎么回事，谷歌的招聘委员会是如何运作的。是的，每家公司各具特色。了解这些“怪癖”会让你更加胸有成竹，不会被突如其来的亚马逊“调杆员”给吓住，也不会对苹果居然同时派出两位面试官来考察你而感到意外。

---

① “bar raiser”（调杆员）的概念来自亚马逊美国总部。这个词原指在跳高比赛中，一次次将杆调高的工作人员。而亚马逊的调杆员则是一群在招聘过程中负责从企业文化以及行为准则的角度考察应聘者，从而维护招聘质量的人。在招聘中，调杆员会用很苛刻的眼光考察应聘者是否在至少一点上高过亚马逊的平均水准，如果是，那么雇用这样的人实际上就等于在提升公司的能力，这就起到了“抬杆”的作用。——编者注

此外，这些专家也强调了各家公司的面试重点。尽管这些顶尖公司都喜欢考察求职者的编码能力和算法基础，他们其实也各有侧重。不管这是源自各家公司的技术背景或是历史，至少你知道该如何做好准备。

接下来，让我们一起揭开微软、亚马逊、谷歌、苹果、Facebook和雅虎的“面试迷局”吧。

2.1　微软面试

微软喜欢招聪明人，尤其青睐计算机极客。求职者必须对技术满怀热情。微软的面试官不大会问你一些C++ API的个中细节，而是直接让你在白板上写代码。

参加面试时，求职者最好在早上约定时间之前赶到微软，先填好一些表格。接着你会和招聘助理碰面，他会给你一个面试样题。招聘助理主要是帮你热热身，不大会问技术问题；就算真的问了几个简单的技术问题，也是想让你放松心情，等到面试真正开始时，你就不会那么紧张了。

对招聘助理一定要以礼相待。说不定他们会帮上大忙，在你首轮面试表现欠佳时，他们有可能帮你争取重新面试的机会。夸张地说，他们甚至还能左右你的应聘结果。

面试当天你会接受4～5轮面试，面试官一般来自两个团队。许多公司会把面试安排在会议室，而微软的面试一般在面试官的办公室进行。你正好可以借机四处看看，感受一下他们的团队文化。

一轮面试过后，不同的团队，做法不一样，面试官可能会根据个人习惯决定是否将你的表现反馈给后续的面试。

完成所有面试后，你有可能会见到招聘经理。假如真是这样的话，那可是好兆头，这意味着你通过了某个团队的基本考察。接下来，就要看招聘经理要不要录用你了。

快的话，面试当天你就会知道结果，慢的话，则可能要等上一周。要是等了一周还没收到人事部的通知，不妨发封邮件，客气地问一下进展。

如果你没有马上收到回应，有可能是因为招聘助理太忙了，这并不代表你就没戏。

必要准备事项

“你为什么想要加入微软？”

提这个问题，微软是想了解你是否对技术满怀热情。一个比较好的答案是：“自打接触计算机以来，我就一直在用微软的软件，贵公司开发的软件产品令人赞不绝口。比如，我最近一直在Visual Studio开发环境中学习游戏编程，它的API实在是太好用了。”注意这个答案是如何展示你对技术满怀热情的。

独特之处

如果到了招聘经理这一关，说明你面试表现得不错。这可是个好兆头！

2.2　亚马逊面试

亚马逊的招聘流程一般从两轮电话面试开始，期间求职者会接受某个团队的面试。偶尔也会出现面试3轮甚至更多轮的情况，可能是有位面试官对你的评价不高，或是别的团队对你有兴趣。此外，还有其他特殊情况，比如求职者就在亚马逊总部所在地西雅图，或他以前面试过其他职位，也许一次电话面试就够了。

在电话面试中，面试你的工程师通常会要求你通过共享文档工具（如CollabEdit）写些简单的代码。他们问的技术问题可谓五花八门，意在探测你究竟熟悉哪些领域。

接下来，如有一两个团队根据你的简历和在电话面试中的表现相中你，你就要飞到西雅图接受4～5轮面试。在白板上写代码是少不了的，有些面试官会着重考察你的其他技能。每一轮面试官都会侧重不同的领域，所以他们的提问会大相径庭。在提交自己的评价报告之前，他们看不到其他面试官对你的评价，而且公司也不鼓励面试官在面试过程中互相交流，一切讨论都得等到几轮面试全部结束后。

顾名思义，“调杆员”主要负责把控面试质量。他们受过专门训练，并且是从其他团队抽调来的，以减少面试中的主观倾向。在面试中，如果有位面试官风格迥异且要求格外严格，那他可能就是传说中的“调杆员”。这种人不仅面试经验丰富，而且跟招聘经理一样，拥有生杀大权。不过，切记：这一轮面试表现磕磕绊绊，并不等于你的整体表现就很差。面试官会比照其他求职者来评价你的水平，而不是只看你答对多少问题。

等到所有面试官提交评价报告后，他们会在一起讨论你的表现，并决定是否录用你。

一般来说，亚马逊的招聘团队都会很快给出录用结果，很少有耽搁。要是一周内都没等到结果，建议你发封措辞得当的邮件询问进展。

必要准备事项

亚马逊是一家互联网公司，这也意味着他们非常关注“扩展性”问题。请做好相应的准备。当然，回答这些问题，并不要求你具备分布式系统方面的知识。具体建议可参看“扩展性与存储限制”一节。

此外，亚马逊还会问很多面向对象设计的问题。请参看“面向对象设计”一节，里面有一些样题和建议。

独特之处

“调杆员”来自其他团队，旨在提高面试标准。他和招聘经理一样重要，请尽量表现得出色一些。

2.3 谷歌面试

业界流传着很多有关谷歌面试的可怕谣传，但多数也只是谣传。谷歌的面试与微软或亚马逊的并无太大区别。

谷歌的面试也从电话面试开始，来面试你的人是技术工程师，因此免不了会问些技术难题，求职者切不可掉以轻心。这些问题也可能涉及编程，有时你还要通过共享文档工具写些代码。电话面试的问题和现场面试的类似，要求也一样。

现场面试一般有4～6轮，其中一轮为午餐面试。面试官之间不能透露自己的评价报告，因此每一轮面试你都可以从零开始。午餐面试不会有评价报告，你可以借机问些其他环节不方便问的问题。

谷歌不会要求面试官侧重不同的领域，也没有所谓的标准流程或结构。每个面试官可以自行决定问哪些问题。

面试过后，评价报告会以书面形式提交给由工程师和经理组成的“招聘委员会”，由他们作

出录用结论。面试评价报告由分析能力、编程水平、工作经验和沟通能力等四部分组成，最后你会得到总的评分，在1.0到4.0之间。“招聘委员会”里一般不会有你的面试官。就算有，那也纯属巧合。

通常，在决定录用与否时，招聘委员会更看重那种有面试官给你打高分的情况，打个比方，如果你的得分是3.6、3.1、3.1 和2.6，效果要好过拿4个3.1。

也就是说，每轮面试不一定都要有上佳表现。此外，你在电话面试中的表现一般起不了决定性作用。

如果招聘委员会给出的意见是“聘用”，你的材料就会转给“薪酬委员会”及“执行管理委员会”。最终结果可能要等上几周，因为还有不少流程要走，等待多个委员会审批。

必要准备事项

作为一家互联网公司，谷歌非常看重如何设计可扩展的系统。因此，务必掌握“扩展性与存储限制”一节的问题。此外，谷歌的面试官很喜欢问些涉及“位操作”的问题，也请重点复习这些方面的知识。

独特之处

面试官不是决策者。他们只提交评价意见供招聘委员会参考。招聘委员会给出录用与否的决定，当然，该决定偶尔也会被谷歌高管否决。

2.4　苹果面试

苹果的面试流程与公司本身的风格非常相符，是最没官僚味儿的。苹果的面试官很看重技术功底，但求职者对应聘职位和公司的热情也非常重要。虽然成为Mac用户并不是应聘苹果的先决条件，但你至少要对该系统有一定了解。

在苹果的面试流程中，招聘助理会先给你打电话了解一些基本情况，接下来团队成员会对你进行一连串的技术电话面试。

当你受邀去参加现场面试时，招聘助理会出面接待你，并介绍面试的大致流程。然后，你要接受招聘团队成员6～8轮的面试，其中这个团队的重要人物也会来面试你。

苹果的面试形式是一对一或二对一。请做好在白板上写代码的准备，交流的时候一定要把自己的思路表达清楚。你可能会跟未来的上司共进午餐，这看似随意，但其实也是一次面试。每个面试官都会侧重不同的领域，面试官之间一般不会过问彼此的面试情况，除非他们想让后续面试官就求职者某一方面多挖掘点内容。

当天所有面试结束后，面试官会在一起商议你

必要准备事项

如果你知道哪个团队会来面试你，务必先熟悉他们的产品。你喜欢该产品的哪些方面？你觉得有哪些可以改进的地方？给出独到见解可以有力展示你对这份工作的激情。

独特之处

在苹果的面试中，二对一的形式司空见惯，不过也不用太紧张——这跟一对一面试并无分别。

此外，苹果的员工都是超级果粉，在面试中，你最好也能展现出同样的热情。

的表现。如果大家都认为你表现不错，接下来会由你应聘部门的主管或副总来面试你。能见到主管也不见得你一定会被录用，不过总归是个好兆头。让不让你见主管的决定对你是不公开的，如果你落选了，他们只是默默送你离开公司，也不会透露你为什么落选了。

如果你得以进入主管或副总面试环节，面过你的面试官会聚到会议室正式表决录用意见。副总通常不会列席，但如果你没能打动他们，他们照样可以直接否决。招聘人员通常会在几天后联系你，要是等不及的话，你也可以主动联系。

2.5 Facebook 面试

Facebook的在线工程难题[1]曾引发热议，其实这无非又是吸引眼球的手段之一。除了解答这些难题，你还可以通过传统渠道申请该公司的职位，比如提交在线职位申请，或者参加校园招聘会。

一旦被Facebook挑中，求职者一般至少要接受两轮电话面试。不过，公司所在地[2]的求职者可以少一轮。电话面试主要涉及技术问题，求职者通常要用Etherpad或其他共享文档工具写些代码。

如果你还在上学，在学校接受面试，那你还要写代码。面试官会要求你在白板或白纸上写代码。

现场面试时，主要由其他软件工程师来面试你，不过，招聘经理有空的话也会参与。所有面试官都受过专业面试培训，他们只提供意见，对你的应聘结果不作决断。

现场面试的每个面试官都各有侧重，以确保大家不会重复提问，并全面考察求职者的能力水平。面试问题主要分为算法、编程水平、软件架构/设计能力等几大块，同时，面试官也会考察你能否适应Facebook快节奏的开发环境。

面试过后，在交流你的表现之前，面过你的面试官会先提交书面评价报告。这么做是为了确保各位面试官能对你的表现作出相对独立的评价。

一旦收到所有的评价报告，面试小组和招聘经理便会商讨你的面试结果。他们会先达成统一意见，然后提交给招聘委员会。

Facebook很看重“忍术”（灵活应变）——也就是使用任何语言快速构建优雅、可扩展解决方案的能力。懂PHP并不会显得特别突出，因为Facebook也有很多后台工作要用到C++、Python、Erlang和其他语言。

必要准备事项

作为网络科技的新贵及“当红炸子鸡”，Facebook也更青睐那些富有创业精神的开发人员。在面试过程中，你要展现出自己热衷创造新事物的激情。

独特之处

Facebook由公司统一招聘员工，而不是专门针对某个团队。面试成功并入职后，你会先参加为期6周的“新兵训练营”，帮你快速适应大规模的代码库。资深工程师会担任你的导师，辅导你掌握最佳实践和必备技能，最终让你可以游刃有余地加入自己喜欢的项目组。

---

① 感兴趣的读者可以访问页面Facebook Engineering Puzzles：www.facebook.com/careers/puzzles.php。——译者注

② Facebook总部位于美国加利福尼亚州的门罗帕克市，地址为黑客路1号（1 Hacker Way）。——译者注

2.6 雅虎面试

雅虎往往只招美国排名前20的高校毕业生，不过其他求职者仍可通过雅虎公开招聘渠道（或者，可以内部推荐的话就更好了）得到面试机会。取得面试资格后，你会先接受一轮电话面试。对你进行电话面试的一般是资深员工，比如技术主管或经理。

在现场面试中，一般由来自同一团队的六七个人来面试你，每轮面试时长45分钟。每个面试官都会侧重不同的领域。比如，有的面试官可能侧重于数据库知识，而有的面试官则会关注你对计算机体系结构的理解。每轮面试的时间安排大致如下。

- **开头5分钟：**一般对话。比如，自我介绍，聊聊项目经历等。
- **中间20分钟：**编程问题。比如，实现归并排序。
- **最后20分钟：**系统设计问题。比如，设计一个大型分布式缓存系统。这些问题往往与你以往的项目经历或面试官当前在做的工作有关。

当天面试结束后，你可能还会跟项目经理或其他人面谈一次。内容包括产品展示、你对雅虎的疑虑以及你手上有无其他公司的录用通知，等等。这次面谈旨在增进双方了解，通常不会影响你的面试结果。

与此同时，之前的面试官会讨论你的表现并尝试作出结论。最终录用与否由招聘经理决定，他会综合考虑面试官对你的正面及负面评价。

如果你的表现不错，有可能当天就会收到口头录用通知。但也不一定。也许他们要过几天才通知你，个中原因不一，比如，你应聘的团队可能还想再面试几个人看看。

必要准备事项

雅虎面试少不了系统设计问题，几乎成了惯例，所以，还请做好相应的准备。他们想要确认你不仅会写代码，而且还能设计软件。要是没有这方面的知识，也不要紧，你仍然可以给出自己的设计思路。

独特之处

雅虎的电话面试一般由拥有决定权的人负责，比如招聘经理。此外，雅虎往往会在当天给出面试结果（如果你能入他们法眼），这一点很特别。在你进行最后一轮面试的同时，其他面试官也正在讨论你的表现。

第 3 章

特殊情况

- 有工作经验的求职者
- 测试人员及SDET
- 项目经理与产品经理
- 技术主管与部门经理
- 创业公司的面试

3.1　有工作经验的求职者

只要你仔细读过之前的章节，遇到以下情况应该也不会太惊讶：在面试中，对于有工作经验的求职者和初出茅庐的新手，面试官会问同样的问题，而且面试标准差别也不大。

你可能知道，大多数面试题都是些涉及数据结构与算法的常见问题。多数公司认为这是检验个人能力的上佳手段，故而对所有求职者一视同仁。

有些面试官可能会对有工作经验的求职者稍稍提高标准和要求。毕竟，他们有多年的工作经验，理应比新手表现得更出色，不是吗?

不过，也有一些面试官持相反的看法。有工作经验的求职者离开学校太久了，可能毕业后就没怎么接触过这些基本概念。他们忘记其中一些细节也在情理之中，所以我们应该稍微降低标准。

总体来看，两者相抵。所以，如果你是有工作经验的求职者，碰到的问题和面试标准基本上与新手相差无几。不同之处在于系统设计和架构方面以及与你简历相关的问题。

一般来说，学生在系统架构方面没有什么积累，这类经验只有通过实践才能获得。因此，面试官会根据你的经验水平来评估你在这些问题上的表现。当然，在校生和应届毕业生也会被问及这方面的问题，总之都要竭尽全力做好准备。

此外，对于“说说你碰到过的最棘手的bug？”之类的问题，面试官往往期待有工作经验者给出更加深入、让人印象深刻的答案。你拥有更丰富的经验，回答自当不同凡响。

3.2　测试人员及 SDET

软件开发测试工程师（SDET）这个职位确实比较复杂。作为SDET，不仅要写得一手好代码，还得是优秀的测试人员。

建议大家从以下几点入手准备SDET的面试。

- **准备核心测试问题**：例如，怎么测试一只灯泡？一支笔？一台收银机？抑或是微软的Word软件？参看本书“测试”一节，有助于你在这些问题上准备得更充分。
- **练习编程问题**：应聘SDET被拒的最大原因就是编程能力不足。尽管这个职位对编程能力的要求比SDE（软件开发工程师）略低，但面试官还是期待SDET具备很强的编程能力和算法功底。准备过程中，不妨拿针对普通开发人员的编程和算法题来练手。
- **练习测试编码问题**：对SDET来说，这类问题的常见问法是“写代码实现X功能”，紧接着就是，“好，请测试你写的代码”。就算面试官没有提这个要求，你也应该问问自己：“我该如何测试这段代码？”切记：SDET可能碰到任何问题！

对测试人员来说，具备良好的沟通能力也非常重要，因为这份工作要求你跟各种各样的人打交道。因此，不要对行为面试题掉以轻心，可参看“行为面试题”一章。

职业生涯建议

最后，提几点职业生涯建议：如果你跟许多求职者一样，认为应聘SDET职位是进入一家公司的“捷径”，那就必须想清楚，从SDET转开发岗位可不轻松。假如你有此意图，务必加强自己的编程能力和算法功底，并尽可能在一两年内转岗。否则，“温水煮青蛙”，拖得越久，你的目标就越难以实现。

总之，常写代码，以防手生。

3.3　项目经理与产品经理

不同公司的PM职位大相径庭，甚至在同一家公司都可能大不相同。例如，微软有些PM职位其实相当于“口碑传道者”，职责是面向客户推广公司产品，有点接近市场营销。然而，微软内部的其他PM则可能每天要花大量时间编写代码。后一种PM在面试中很可能会被问到编码问题，因为这是其工作职责的重要部分。

大体上，求职者应聘PM职位时，面试官主要考察以下几个方面。

- **处理含糊情况**：虽然它不是面试中最重要的考察面，但你要明白面试官的确很看重此技能。他们想看到你面对含糊情况不会手忙脚乱、不知所措；希望看到你迎难而上，比如寻找新的信息、优先考虑最重要的模块，并以有条理的方式解决问题。面试官一般不会直接考察你这方面的能力（但也不排除这种可能性），不过他们可能会根据你在处理问题时的表现对你进行评估。
- **以客户为中心（态度层面）**：面试官希望看到你能做到以客户为中心。你是会照搬自己的经验主观臆测客户使用产品的方式，还是会站在客户的立场来了解他们希望如何使用产品？诸如“为盲人设计一款闹钟”的面试题考查的正是这个方面。当你听到这类面试题时，务必多提问题以了解产品主要面向**哪些客户**，以及他们会**如何**使用该产品。本书“测试”一节有很多相关内容可供参考。

- **以客户为中心（技术层面）**：有些团队做的产品功能非常复杂，要求PM求职者必须充分掌握相关产品，因为等到工作时再上手是来不及的。欲在MSN Messenger团队中谋得PM一职，也许不一定要精通即时通讯工具，而从事Windows Security工作则可能要求你具备扎实的计算机安全功底。因此，除非掌握了必备技能，否则在面试之前你还是三思而后行吧！
- **多层次交流能力**：PM需要跟公司内各个级别、跨部门跨职能人士打交道。所以，面试官会希望你具备多层次交流能力。这方面的考查非常直接，比如，面试官会抛出类似“向你的祖母解释什么叫TCP/IP”的问题。当然，从你如何描述此前的项目经历，他们也能看出你的沟通能力。
- **对技术的热情**：快乐工作的员工往往是高产员工，所以公司要确保你喜欢并享受这份工作。在你的回答中，应该处处展示自己对技术的热情，同时，要是能对公司或团队充满热情就更好了。面试官可能会直接问你：“为什么想来微软工作？”此外，他们也乐于见到你充满激情地描述自己此前的工作经历和遇到过的挑战。面试官喜欢那些不惧挑战并迎难而上的求职者。
- **团队合作/领导能力**：这大概是PM面试中最重要的方面，无疑也是这份工作本身的关键所在。所有面试官都会评估你能否与其他人合作无间。他们常会提出这类问题：“说说你怎么处理团队成员没能按进度完成工作的情况。”此外，面试官也想了解你能否妥善处理冲突、是否积极主动、是否了解你身边的人，以及人们喜不喜欢与你共事。你在“行为问题”上所做的准备在这里就显得尤为重要。

以上这些方面都是PM的必备技能，因此也是面试的重点。各个方面的权重大致取决于你应聘的PM职位以及该职位具体看重哪些方面。

3.4 技术主管与部门经理

基本上，技术主管职位都要求具备很强的编程技能，部门经理职位往往也不例外。如果这份工作需要编写代码，那你就必须具备很强的编码技能和算法功底——要求不比普通开发人员低。特别是谷歌，在编程上，对部门经理的要求很高。

此外，你还要做好以下准备。

- **团队合作/领导能力**：任何担任管理类角色的人都必须懂得团队合作，并能领导员工。面试官会或明或暗地考察你是否具备这些能力。一方面，他们会直接询问你在此前工作中是如何处理冲突的，比如你与主管意见相左的时候；另一方面，面试官也会暗中观察你怎么与他们互动。如果你的态度过于傲慢或太顺从，那他们就会认为你不太适合当管理人员。
- **把握轻重缓急**：管理人员经常要面对层出不穷的状况，比如怎样才能确保团队在即将到来的截止期前完成工作。你需要充分展示你在一个项目中分得清轻重缓急，砍掉无足轻重的部分。把握轻重缓急意味着要通过正确的提问来掌握哪些方面至关重要，以及合理预估出都能实现哪些方面。

- **沟通能力**：管理人员不仅需要与上下级沟通，而且可能还会与客户或其他不太懂技术的人进行交流。面试官希望看到你具备与各种人打交道的能力，跟他们沟通起来游刃有余。实际上，面试官也是在拐弯抹角地评估你的个性。
- **“把事情做好”的能力**：经理与主管最重要的职责也许就是“把事情做好”。这意味着你要在项目准备和具体实施之间达成适当的平衡。你需要掌握如何组织项目，以及如何激励员工，从而达成团队目标。

最终，这些方面大都会跟你的过往经验和个性关联起来。务必利用“面试准备表格”做好充分准备。

3.5　创业公司的面试

创业公司（start-up）的职位申请和面试流程千差万别。我们没办法述及每一家创业公司的情况，好在还能列举一些共通之处。不过，也请理解，实际情况可能会有所不同。

1. 职位申请

很多创业公司都会在网上发布招聘启事，但对于那些最热门的创业公司，最好的申请方式是通过内部推荐。这个推荐人不必非得是你的密友或同事。你可以四处撒网，向认识的人表达自己的意向，然后也许有人会拿起你的简历看看你是不是合适人选。

2. 签证与工作许可

很遗憾，美国大多数小型创业公司没有能力为你申请工作签证。他们跟你一样痛恨劳工部教条的制度，可还是无能为力。如果你没有合法身份，同时又想到创业公司工作，也许最好的选择就是找一家为创业公司输送人才的专业人力资源代理机构，又或者，你可以盯着那些规模较大的初创公司。

3. 简历筛选因素

创业公司需要的工程师不仅要聪明过人，会写代码，而且同时也能在创业环境中卖力地工作。你的简历应该展示这些特质。

此外，你还必须充满干劲，积极做到最好；这些创业公司急需立马能上手干活的员工。

4. 面试流程

与大公司注重你在软件开发上的整体职业素养相比，创业公司更注重你的个性契合度、技术技能和此前的工作经验。

- **个性契合度**：面试官会通过你与他们的互动来评估你的个性契合度。请注意，与面试官交流时要友善、专注，这会给人留下好印象，从而获得更多工作机会。
- **技术技能**：创业公司需要立马能上手干活的人，因此非常看重你在特定编程语言上的能力。如果你恰好掌握该公司使用的编程语言，请务必好好准备与此相关的各种细节问题。
- **以往经验**：创业公司会问你很多以往工作经验有关的问题，请特别关注“行为面试题”一章。

除此之外，你还会碰到这本书中提及的很多编程及算法问题。

第 4 章

面试之前

- 积累相关经验
- 构建人际网络
- 写好简历

4.1 积累相关经验

录用与否主要取决于你在面试中的表现，而简历和过往经验则决定你有没有面试机会。你应该想方设法提升自己的技术（及非技术）水平。不管是应届毕业生还是专业人士，拥有额外的编程经验都会让你受益匪浅。

在校生可以采取下面这些举措。

- **选修有大作业的课程**：如果你还是学生，请不要避开那些有大作业的课程。将来，你可以把这些项目经历都写在简历上，这会大幅提高得到顶尖科技公司面试机会的几率。当然，这些项目与实际情况联系越紧密，效果就越好。
- **找一些实习生工作**：就算你是大学新生，也有机会得到相关的专业经验。大一、大二的学生可以考虑参加诸如“微软探索者”和“谷歌编程夏令营”这样的活动。如果得不到类似的机会，进入创业公司历练一下也不错。
- **开拓一些业务或项目**：绝大多数公司都青睐富有创业精神的人。此举不仅可以培养一些技术经验，而且同时也能展示你的主观能动性和把事情做好的能力。你可以利用周末和休息时间写个软件。要是认识学校教授，不妨试着请他予以“资助”，以便你将自己的工作变成一项独立研究。

另一方面，专业人士可能早已累积好相应资本，准备跳槽进入他们梦寐以求的公司。比如，谷歌的开发人员可能已经攒够经验，有机会跳槽到Facebook工作。不过，如果你想从不知名的小公司跳到科技巨头公司，或者从测试岗位转成开发人员，请参考以下这些建议。

- **多承担一些编程职责**：在不透露跳槽意向的前提下，你可以向经理表达自己想在编程上接受更大的挑战。尽可能地参与一些重大项目，并多多使用对自己以后有利的技术，将来它们会成为简历上的亮点。另外，简历上也要尽量多列举这些与编程相关的项目。

- **善用晚上和周末的闲暇时光**：如有空闲时间，可以试着构建一些手机应用、网页应用或桌面软件。这样，你就有机会接触到时下流行的新技术，从而更契合科技公司的要求。这些项目经验都可以写到简历上，没有什么比"为兴趣而工作"更能打动招聘人员的了。

总而言之，公司最青睐的人才必须具备两大特性：一是天资聪颖，二是扎实的编程功底。要是你能在简历上充分展示这两点，面试机会就唾手可得了。

此外，你应当提前规划好职业发展路径。如果打算转型成为管理者，哪怕当下应聘的仍是开发岗位，也应现在就想方设法培养自己的领导才能。

4.2 构建人际网络

你或许听说过很多人靠朋友推荐找到了好工作。不过，你可能想不到，还有更多人是通过*朋友的朋友*找到工作的。这真的很有道理。用极客的话来说，你有N个朋友，也就意味着你有N^2个朋友的朋友。

那么，在你找工作时这个数字意味着什么呢？这意味着，不管是直接联系人还是拐弯抹角的关系，对你找工作都很有帮助。

1. 什么叫好的人际网络

好的人际网络不仅意味着你广交朋友（广度），还要与他们保持紧密的联系（深度）。这句话看似矛盾，实则要辩证地看待。

- **广度**：你的人际网络中不仅要有业内技术人士，而且最好还能涵盖各行各业的人才。比如说，结交一位会计朋友会对你的职业生涯帮助很大，因为他很可能在其他领域有很多朋友。有时候，其中有些人可能就想认识像你这样的技术人才。请抱着开放的交友态度去对待他人。
- **深度**：通过自己的密友来结交新朋友是个不错的方法，总好过让不太熟的人为你牵线搭桥。此外，人们会对那些所谓的"老油条"和"交际花"避之唯恐不及，觉得这些人太虚伪了。因此，尽量与朋友保持真诚和深厚的关系。

其中的微妙之处就在于找到平衡点，你认识的人当然越多越好，但要确保自己待人真诚、开放。如果只是热衷于收集大家的名片，那你最终往往只会一无所获。

2. 如何构建坚实的人际网络

有些人认为，我们应当走出家门，去结识更多人。这么说也有道理。但是去哪里呢？而且，如何才能将"点头之交"发展成好朋友呢？

以下这些建议或许能给你一些启发。

(1) 通过Meetup.com这样的社交网站或校友网来获取你感兴趣的活动资讯。记得带上你的名片。如果你暂时没有工作或还是学生，那就自己印些名片。

(2) 主动跟人打招呼。也许你生性胆怯，不敢迈出这第一步。但请相信我，没人会拒绝你的友好之举，甚至有些人还会欣赏你的自信。话说回来，最坏能坏到什么地步呢？他们不喜欢你，不会与你结交，从此和你老死不相往来吗？

(3) 大大方方地聊你的兴趣，并和人们谈论他们的兴趣。如果他们正在运营创业公司，或是从事其他你也感兴趣的活动，不妨邀请他们一起喝咖啡继续畅谈。

(4) 活动结束后，你可以在LinkedIn上把他们加为好友，或者给他们发邮件。当然，更好的方式就是邀请他们一起喝咖啡，这样你们就会有充足的时间来畅谈他们的创业公司，或是双方都感兴趣的话题。

(5) 最重要的是乐于助人。经常助人一臂之力，你就会给人留下慷慨大方、友好和善的印象。那些乐善好施的人往往也会得到更多帮助。

切记，不要只局限于现实生活中的社交。在这个信息爆炸的时代，社交还可以拓展到网络上，通过博客、微博、Facebook和电子邮件结交朋友。

当然，也不要太“走火入魔”沉迷于在线社交，你得努力建立实实在在的人际关系。

4.3 写好简历

简历筛选标准与面试标准并无太大差别，也是看你是否又聪明又会写程序。

这意味着你在准备简历时应该突出这两点。提到自己喜欢打网球、旅游或玩魔法牌可没什么用。在罗列这类无关紧要的爱好之前，务请三思，宝贵的篇幅应该用来展示自己的技术才能。

1. 简历篇幅长度适中

在美国，人们会建议工作经验不足10年的求职者将简历压缩成一页；超过10年的，至多用两页。为什么呢？主要有两大理由。

- 招聘人员浏览一份简历一般只会用20秒钟左右。要是你的简历言简意赅恰到好处，招聘人员一眼就能看到。废话连篇只会模糊重点，扰乱招聘人员的注意力。
- 有些人遇上冗长的简历连看都不看。你真的想冒这个风险，让别人直接扔掉你的简历吗？

如果看到这里你还在想，我工作经验太丰富了，一页篇幅根本放不下怎么办？相信我，你可以的。一开始大家都会这么说。其实，简历写得洋洋洒洒并不代表你经验丰富，反而只会显得你完全抓不住重点。

2. 工作经历

简历不是也不应该是关于工作经历的编年史。比如，卖过冰淇淋跟聪明与否或代码写得怎么样关系不大。你应该只列举那些相关的工作经验。

● 列举要点

在描述工作经历时，请尽量采用这样的格式：“使用Y实现了X，从而达到了Z效果。”比如，下面这个例子：

- “通过实施分布式缓存功能减少了75%的对象渲染时间，从而使得用户登录速度加快了10%。”

下面还有一个例子，描述略有不同：

- “实现了一种新的基于windiff的比较算法，系统平均匹配精度由1.2提升至1.5。”

尽管不是所有经历都能套用此句型，但原则无二：描述做过的事情、怎么做的，以及结果如何。理想的做法是尽可能地量化结果。

3. 项目经历

在简历中列出“项目经历”这一部分会让你看起来很专业。对于大学生和毕业不久的新人尤其如此。

简历上应该只列举2到4个最重要的项目。描述项目要简明扼要，比如使用哪些语言和技术。你也可以加上一些细节，比如该项目是个人独立开发还是团队合作的成果，是某一门课程的一部分还是自主开发的。当然，这些细节不一定放到简历上，除非能让简历更出彩。

项目也不要列太多。很多求职者就犯过这样的错误，在简历上一股脑儿列出先前做过的13个项目，鱼龙混杂，效果反而不佳。

4. 编程语言和软件

- 软件

一般说来，“熟悉微软Office”之类不必列入简历。这应该是地球人的必备技能，列出来反而会模糊重点。你应该列出那些能反映自身技术水平的软件或系统（比如Visual Studio、Linux等），不过坦白说，这么做用处也不大。

- 编程语言

列举编程语言确实是件难事。我们到底应该列出自己用过的所有语言，还是只列那些用得最顺手的语言呢？我建议采用下面这个折中办法：列出你用过的主要语言，后面加上熟练程度。比如像下面这样：

- 编程语言：Java（非常熟练），C++（熟练），JavaScript（有过使用经验）。

5. 给母语为非英语的人及国际人士的建议

一些公司可能会因为小小的笔误就扔掉你的简历，所以请至少找一位以英语为母语的人来帮你审阅简历。

此外，申请美国的工作时，简历中不要包含年龄、婚姻状况或国籍等。公司并不想看到这些个人信息，因为怕惹上不必要的麻烦。

第 5 章

行为面试题

- 准备工作
- 如何应对

5.1　准备工作

行为面试题的考察有各种各样的原因。人们可以通过这些问题来了解你的个性，或是更深入地掌握你的履历，又或者缓和一下面试的紧张气氛。不管怎样，这个部分很重要，而且有办法做好准备、有的放矢。

准备工作

行为面试题一般是这么问的："说说你曾经……"面试官可能还会要求你列举并说明具体的项目或岗位。我建议你先按如下格式拟定一份"准备表格"：

常见问题	项目1	项目2	项目3	项目4
最难的部分				
有什么收获				
最有意思的部分				
最难解的bug				
最享受的过程				
与团队成员的冲突				

第一行可以列举你在简历中提到的主要事项，比如项目、职位或活动。第一列应该写一些常见问题：你最享受和最不喜欢的过程、最难的部分、从中学到的经验、最难解的bug，等等。然后，在对应单元格里写下相应的小故事。

当面试官问及项目有关的问题时，你就能回想起这些小故事，从容应对。记得在面试前复习这份表格。

另外，建议大家将小故事浓缩成几个关键字，以便填到单元格里。这样一来，这份表格用起来就会更顺手，方便记忆。

电话面试时，最好将这份表格摆在自己跟前。把每个小故事都概括成几个关键字，更容易记忆，自然而然就能把整个故事串起来，比死记硬背一段文字要轻松得多。

你还可以将这份表格扩展成一系列“软问题”，比如团队冲突、项目失败的经历以及你需要说服团队成员的事例。对于那些不是专职开发的职位如技术主管、PM或测试人员而言，这些都是很常见的面试问题。如果你刚好要申请其中一个职位，建议你针对这些“软问题”再准备一份表格。

在回答问题时，你不只是在讲述一个与该问题密切相关的故事，更是在向别人展现自我。所以，请用心思索每个故事都能体现出自己的哪些特性。

1. 你有哪些缺点

被问及自己有哪些缺点时，回答不要太空泛！诸如“我最大的缺点就是工作太努力了”的回答，反而会显得你傲慢自大，并且不愿正视自己的不足。没有人喜欢与这样的人共事。因此，你应该提到真实、合乎情理的缺点，然后话锋一转，强调自己如何克服这些缺点。比如：“有时候，我可能对细节不够重视。好的一面是我反应迅速、执行力强，但不免会粗心大意而犯错。有鉴于此，我总是会找其他同事帮忙检查自己的工作，确保不出问题。”

2. 项目中最难处理的问题是什么

当面试官问到这个问题时，请不要泛泛地回答“我得学习很多新的编程语言和技术”。除非你实在是无话可说，否则这种回答似乎是在强调：该项目并不是很难，没什么棘手的问题。

3. 你应该问面试官哪些问题

大多数面试官都会给你提问的机会。有意无意间，你提问的质量也会成为他们评估你的整体表现的因素之一。

也许你会在面试过程中临时想到若干问题，但你还是可以并且应该事先准备好问题。对公司和团队做些调研，有助于你准备问题。

问题可以分成以下三大类。

- *真实的问题*

也就是你真的想知道答案的问题。下面是对多数求职者有用的一些问题点。

(1)“你每天有多少时间花在写代码上？”

(2)“你一周要开几次会？”

(3)“整个团队中，测试人员、开发人员和项目经理的比例是多少？他们是如何互动的？团队怎么做项目规划？”

这些问题有助于你较好地了解公司的工作环境和日程安排。

- *有见地的问题*

有见地的问题可以充分反映出你的编程水平和技术功底，同时，还能显示你对该公司或其产品的兴趣。

(1)“我注意到你们使用了X技术，请问你们是如何处理Y问题的？”

(2)“为什么你们的产品选择使用X协议而不是Y协议？据我所知，虽然X有A、B、C等几大好处，但因为存在D问题，很多公司并未采用该协议。”

只有事先对该公司做过充分调研，才问得出这类有深度的问题。

- 富有激情的问题

这些问题旨在展示你对技术的热忱。要让面试官知道你热衷学习，将来能为公司的发展做出很大贡献。比如：

(1)“我对可扩展性很感兴趣。请问你从事过分布式系统方面的工作吗？有哪些机会可以学习这方面的知识？”

(2)“我对X技术不是太熟悉，不过听上去是个不错的解决方案。你能给我多讲讲它的工作原理吗？”

5.2 如何应对

如前所述，面试官喜欢在面试开始和结束时与你谈天说地或聊聊“软技能”。他们通常会就你的简历问些问题，或者泛泛地提问，此时你也可以问一些和公司有关的问题。这个面试环节除了缓和气氛，也是意在了解你。

回答这类问题时，切记以下几个建议。

1. 力求具体，切忌自大

骄傲自大是面试大忌。可是，你又想给面试官留下深刻的印象。那么，怎样才能很好地秀出自己的实力而又不显自大呢？那就是回答问题要具体！

具体也就是只陈述事实，余下的留给面试官自己解读。请看下面这个例子。

- 一号求职者：“我几乎包揽了团队中所有累活和难活。”
- 二号求职者：“我实施了文件系统，因为XXXX等原因，这是整个项目中最难的一部分。”

二号求职者的回答不仅听起来更令人印象深刻，而且也不会显得骄傲自大。

2. 省略细枝末节

当求职者就某个问题喋喋不休时，不熟悉该主题或项目的面试官往往听得一头雾水。所以，请省略细枝末节，只谈重点。换言之，建议你这么回答：“在研究最常见的用户行为并应用Rabin-Karp算法[1]后，我设计了一种新算法，在90%的情况下搜索操作的时间复杂度由$O(n)$降至$O(\log n)$。您要是感兴趣的话，我可以详细说明。”该回答言简意赅，重点突出；要是面试官对实现细节感兴趣，他会主动询问。

3. 回答条理清晰

回答行为面试题有两种常见的组织方式：主题先行法与S.A.R.法[2]。你可以分别或组合使用这两种技巧。

- 主题先行

主题先行即开门见山、直奔主题，回答简洁明了。比如，

① Rabin-Karp算法是由Michael O. Rabin和Richard M. Karp于1987年提出的字符串匹配算法。——译者注

② S.A.R即Situation、Action与Result的缩写，情景、行为与结果。——编者注

- 面试官："讲一讲你必须说服一群人作出大幅调整的事例。"
- 求职者："好的，我在学校提出过一个让本科生互相授课的想法，并成功说服学校采纳该建议。起初我们学校规定……"

主题先行法可以快速抓住面试官的注意力，让他了解事情梗概。此外，假如你有滔滔不绝的倾向，这也有助于你不偏离主题，因为你早已开门见山地点明主旨。

- S.A.R.

S.A.R.法是指先描述情景，然后解释你采取的行动，最后陈述结果。

示例："说说你与某位'刺头'队友相处的事例。"

- **情景**：在操作系统课的大作业中，我被安排与其他三个人合作。其中两人都很卖力，但另外一个人做的不多。开会时他总是沉默寡言，也极少参与邮件讨论，只是很吃力地完成分配给他的模块。
- **行动**：有一天课后，我把他拉到一边讨论这门课程，然后谈起我们的大作业。我坦诚地询问他对大作业的感受，以及他最感兴趣的模块。他建议让他处理最简单的几个模块，并承诺会完成最后的总结报告。我意识到他其实一点都不懒——他只是对这项大作业感到很困惑，并且缺少自信心。此后，我开始与他合作，进一步细分组件模块。此外，在工作中我还经常称赞他以增强他的自信心。
- **结果**：他依然是我们团队最弱的一员，但是进步很大。他及时完成了分配给自己的任务，参与讨论也更积极。后来在另一个大作业中，我们合作得非常愉快。

切记，描述情景与结果务必言简意赅。面试官一般不需要太多细节就知道来龙去脉，实际上，细节过多反而会令他们摸不着头脑。

采用S.A.R.法简明扼要地描述情景、行动和结果，可让面试官快速了解你是如何施加影响的，起到了什么作用。

第 6 章

技术面试题

- 技术准备
- 如何应对
- 算法题的五种解法
- 怎样才算好代码

6.1 技术准备

既然你买了这本书，说明你已经为技术面试做了不少准备。干得好！

即便如此，准备方式也有好有坏。许多求职者只是通读一遍问题和解法，囫囵吞枣。这好比试图单凭看问题和解法就想学会微积分。你得动手练习如何解题。单靠死记硬背效果不彰。

1. 如何练习

就本书的面试题（以及你可能遇到的其他题目），请参照以下几个步骤。

(1) **尽量独立解题**。也就是说，要试着实战演练解题过程。许多题目确实很难，但是没关系，不要怕！此外，解题时还要考虑空间和时间效率。多问问自己 ，能否通过降低空间效率来提高时间效率，或者相反。

(2) **在纸上编写算法代码**。之前你一直在计算机上编写代码，习惯了由此带来的诸多便利。不过，在面试中，你可享受不到语法高亮、代码补全或编译构建的种种好处。不妨在纸上编写代码模拟面试时的情景。

(3) **在纸上测试代码**。也就是要在纸上写下一般用例、基本用例和错误用例等。面试中就得这么做，因此最好提前做好准备。

(4) **将代码照原样输入计算机**。你也许会犯一大堆错误。请整理一份清单，罗列自己犯过的所有错误，这样在真正的面试中才能牢记在心。

此外，模拟面试（mock interview）也非常有用。CareerCup.com提供了与微软、谷歌和亚马逊等公司员工进行模拟面试的机会，当然，你也可以跟朋友一起演练，轮流当面试官给对方做模拟面试。你的朋友不见得受过什么专业训练，但至少还能带你过一遍编码或算法面试题。

2. 你需要掌握的知识

大多数面试官都不会问你二叉树平衡的具体算法或其他复杂算法。老实说，离开学校这么多

年，恐怕他们自己也记不清这些算法了。

一般来说，你只要掌握基本知识即可。下面这份清单列出了必须掌握的知识：

数据结构	算　法	概　念
链表	广度优先搜索	位操作
二叉树	深度优先搜索	单例设计模式
单词查找树（trie）	二分查找	工厂设计模式
栈	归并排序	内存（栈和堆）
队列	快速排序	递归
向量/数组列表	树的插入/查找等	大*O*时间
散列表		

对于上述各项主题，务必掌握它们的具体实现和用法、应用场景、空间和时间复杂度如何等。

对于其中的数据结构和算法，你还要练习如何从无到有地实现。面试官可能会要求你直接实现一种数据结构或算法，或者对其进行修改。不管怎样，你越是熟悉具体实现，把握就越大。

其中，散列表一项特别重要。你会发现，解决面试问题时，经常会用到散列表。

3. 2的幂表

有些人已经把下面这张表背得滚瓜烂熟，如果你还没有的话，面试前一定要背下来。回答可扩展性问题时，这张表用处很大，借助它可以快速算出一组数据占用多少空间。

2的幂	准确值（X）	近似值	X字节转换成MB、GB等
7	128		
8	256		
10	1024	一千	1K
16	65 536		64K
20	1 048 576	一百万	1MB
30	1 073 741 824	十亿	1GB
32	4 294 967 296		4GB
40	1 099 511 627 776	一万亿（trillion）	1TB

有了这张表，就可以做速算。例如，一个将每个32位整数映射为布尔值的散列表可以把一台计算机的内存填满。

在接受互联网公司的电话面试时，不妨将这张表放在跟前，也许能派上用场。

4. 需要知道C++、Java或其他编程语言的细节吗？

我个人不会问这类问题（比如“什么是虚函数表”），不过许多面试官确实会问。

对于微软、谷歌和亚马逊等大公司，我不太担心这些问题。如果你在简历上提到自己熟悉某种语言，那你自然应该掌握这种语言的基本概念。不过，我还是建议你在数据结构和算法方面多下工夫。

参加小公司和非软件公司的面试，这些问题可能更显重要。在CareerCup.com上搜索你心仪的公司再作决定。如果找不到那家公司，那就找一家类似的公司作为参照。一般而言，创业公司更看重与他们使用的编程语言相关的技能。

6.2 如何应对

面试绝非易事。要是没能立刻答出所有问题或某个问题，也没关系！实际上，根据我的经验，在我面试过的120多人中，大概只有10个人能立即答上我经常问的问题。

因此，碰到棘手的问题，不要慌张。只管大声说出你准备怎么解决。向面试官说明你会如何处理这个问题，这样面试官就不会误以为你被难住了。

另外，还有一点：只有面试官点头认可，你才算是真正解决了问题！我指的是，在给出算法后，你就要开始考虑它可能存在的问题。边写代码，边查缺陷。如果你和我面试过的其他110名求职者一样，那就免不了要犯一些错误。

解决技术面试题的五步法

解决技术面试题可采取下面的五步法。

(1) 向面试官提问，以消除疑义。

(2) 设计一种算法。

(3) 先写伪码，但务必告诉面试官接下来会写“真实的”代码。

(4) 写代码要不紧不慢。

(5) 测试写好的代码，仔细修正每一处错误。

下面我们将逐一探讨上述五个步骤。

第一步：提问

技术面试题看似清晰明确实则模糊不清，因此务必多提问题以澄清所有存疑之处。问到最后，你可能会发现，这个问题与你最初预想的截然不同——也许更难，也许更简单。实际上，许多面试官（尤其是微软的）会特意考察你能否提出好问题。

好问题大概是这样的：数据类型是什么？有多少数据？解决这个问题需要什么假定条件？用户都是谁？

示例：“设计一种列表排序算法。”

- 问题：具体是哪种列表？数组还是链表？
  回答：数组。
- 问题：数组里存放的是什么？数字、字符、还是字符串？
  回答：数字。
- 问题：这些数字都是整数吗？
  回答：是的。
- 问题：这些数字来自何处？是身份证号码还是别的什么数值？
  回答：顾客年龄。

- 问题：总共有多少顾客？
  回答：大概一百万。

现在我们要解决一个与最初理解很不一样的问题：对一个包含一百万个整数的数组进行排序，这些整数在0到130（一个合理的最高年龄）之间。该怎么解决这个问题呢？只需创建一个包含130个元素的数组，然后计算每一个元素出现的次数。

第二步：设计算法

算法的设计可能会很难，不过下一节的"算法题的五种解法"可以帮上大忙。在设计算法时，记得问问自己以下几个问题：

- 该算法的空间和时间复杂度如何？
- 碰到大量数据会怎么样？
- 你的设计会引发其他问题吗？例如，你设计了一种二叉查找树的变体，那么该设计是否会影响插入、查找或删除时间？
- 如果还有其他问题或限制，你会做出正确的取舍吗？对于哪些场景，这一取舍可能不是最优的？
- 如果面试官指定特定数据（例如，前面提到待处理数据是年龄值，或按一定顺序排列的），你能否善用该信息？面试官给你特定信息往往是有原因的。

先给出蛮力解法，这么做当然是允许的，甚至推荐这么做。然后，在此基础上不断优化。很显然，面试官总是期望你能给出尽可能最优的解法，但这并不意味着一开始就得给出完美无瑕的答案。

第三步：编写伪码

先写伪码有助于你理清思路，减少犯错的次数。不过，务必先跟面试官打声招呼，你会先写伪码，紧接着就会编写"真实的"代码。许多求职者选择写伪码，意在"逃避"编写真实代码，你肯定不愿与那些求职者为伍。

第四步：编写代码

编写代码不要太仓促；实际上，太仓促很可能会害了你自己。写代码时只管放松步调，做到有条不紊，丝丝入扣。另外，切记以下忠告。

- **多用数据结构**：根据实际情况选用合适的数据结构，或者自己定义数据结构。例如，有个面试题涉及从一群人中找出年龄最小的，不妨考虑定义一个数据结构`Person`表示一个人。这样也能展现出你注重良好的面向对象设计。
- **写代码不要太杂乱**：这看似小事一桩，实则很重要。在白板上写代码时，尽量从左上角而不是中间开始写。这样才有足够的地方从容答题。

第五步：测试

没错，自己写的代码自己测试！考虑测试以下用例。

- 极端用例：0、负数、空值（null）、最大值、最小值。
- 用户错误：用户传入空值或负数会出什么问题？
- 一般用例：测试正常用例。

如果你的算法很复杂或涉及大量数值操作（移位、算术运算等），建议边写代码边测试，而不是写完代码再测试。

发现错误时（这是难免的），务必先弄清楚出现缺陷的原因再作修改。你肯定不希望自己在面试官看来像只热锅上的蚂蚁一样团团乱转，这里修修那里补补。举个例子，我就碰到过这样的求职者，他发现函数碰到某个特定值返回`true`而非正确的`false`，于是直接修改返回值，接着验证函数能否工作。这或许可以修正那种特定情况下出现的问题，但无疑又会滋生新的问题。

当你察觉代码中存在的问题时，务必三思而后改，先理清代码失效的原因。这样你才能写出既漂亮又整洁的代码，也会越写越快。

6.3 算法题的五种解法

要解决棘手的算法问题，世上没有什么不二法门，不过下面介绍的几种方法可能管用。常言道熟能生巧，题目练习得越多，就越容易确定该采用哪种方法来解决问题。

另外，下面这五种方法可以“混搭”使用。也就是说，施以“简化推广法”后，还可以接着尝试“模式匹配法”。

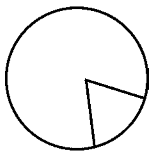
方法一：举例法

我们先从你可能熟悉的“举例法”开始，也许你从未听过这种叫法。“举例法”是先列举一些具体的例子，看看能否发现其中的一般规则。

示例：给定一个具体时间，计算时针与分针之间的角度。

下面以3点27分为例。确定3点的时针位置和27分的分针位置，我们可以画出一个时钟。

在下面的解法中，h表示小时，m表示分钟。同时，我们假定h的范围是0～23。



从这些例子可以得出以下规则。

- 分针的角度（从12点整开始算起）：$360 \times m / 60$；
- 时针的角度（从12点整开始算起）：$360 \times (h \% 12) / 12 + 360 \times (m / 60) \times (1 / 12)$；
- 时针和分针之间的角度：(时针的角度−分针的角度) % 360。

简化上述式子可以得到$(30h - 5.5m) \% 360$。

方法二：模式匹配法

模式匹配法是指将现有问题与相似问题作类比，看看能否通过修改相关问题的解法来解决新问题。

示例：一个有序数组的元素经过循环移动，元素的顺序可能变为“3 4 5 6 7 1 2”。怎样才能找出数组中最小的那个元素？假设数组中的元素各不相同。

这个问题和下面两个问题有点类似：

- 在一个无序数组中找出最小的元素；
- 在一个有序数组中找出某个特定的元素（比如，通过二分查找法）。

- 处理方法

在无序数组中查找最小元素的算法没多大意思（只要遍历所有元素即可），同时它也没有利用给定信息（即这是一个有序数组），因此这个问题帮不上什么忙。

然而，二分查找法就非常适合。我们知道，这是个有序数组，只是一部分元素循环移动过。因此元素排序肯定是从小到大，在某一位置突然变小，接着又开始从小到大排列。那个“转折点”正是最小的元素。

比较中间元素与末尾元素（6和2），由于`MID > RIGHT`，可以确定这个转折点就在这两个元素之间。这不符合从小到大的排列顺序，故而表明转折点就在其中。

如果`MID`比`RIGHT`小，则说明转折点要么在前半部分，要么根本不存在（此数组严格按照从小到大排序）。不管怎样，我们都可以找到最小的元素。

我们可以继续运用这个方法，将数组逐步二分进行查找，最终找到最小的元素（或是转折点）。

方法三：简化推广法

采用简化推广法，我们会分多步走。首先，我们会修改某个约束条件，比如数据类型或数据量，从而简化这个问题。接着，我们转而处理这个问题的简化版本。最后，一旦找到解决简化版问题的算法，我们就可以基于这个问题进行推广，并试着调整简化版问题的解决方案，让它适用于这个问题的复杂版本。

示例：从一本杂志里剪下一些单词可以拼凑成一封勒索信。怎样才能断定勒索信（以字符串表示）是否由某本杂志（即另一个字符串）里的单词组成？

我们可以先这样简化问题：暂时不考虑单词，只当它是字符。也就是说，假设我们从杂志里剪下一些字符拼成了这封勒索信。

接着，我们只需新建一个数组并数出字符的数量，即可解决这个简化后的勒索信问题。数组中的每个元素对应一个字母。首先，我们数出每个字符在勒索信中出现的次数，然后再遍历整本杂志，确认它是否包含勒索信上的全部字符。

推广这个算法时，具体做法和上面的差不多。只不过这一回，我们不再创建包含字符计数的数组，而是创建一个散列表，将单词映射到其词频上。

方法四：简单构造法

对于某些类型的问题，简单构造法非常奏效。使用简单构造法，我们会先从最基本的情况（比如$n = 1$）来解决问题，一般只需记下正确的结果。得到$n = 1$的结果后，接着设法解决$n = 2$的情况。接下来，有了$n = 1$和$n = 2$的结果，我们就可以试着解决$n = 3$的情况了。

最后，你会发现这其实就是一种递归算法——知道$N-1$时的正确结果，就能计算出N时的结果。有时，只有等到算出N为3或4时的结果，我们才能从中找到规律，基于前面的结果解决整个问题。

示例：设计一种算法，打印某个字符串所有可能的排列组合。为简单起见，假设字符串中没有重复字符。

以字符串abcdefg为例：

只有“a”的情况， 结果为：{“a”}

然后是“ab”， 结果为：{“ab”，“ba”}

再然后是“abc”，结果会是什么呢？

此时，问题开始变得“有点意思”了。得到P(“ab”)的答案，怎么才能生成P(“abc”)呢？很简单，新字符是“c”，我们只需在前一种情况的答案也即字符组合的任意位置加一个c就可以了。也就是：

P(“abc”) = 将“c”字符插入P(“ab”)得到的所有字符串的任意位置。

亦即：P(“abc”) = 将“c”字符插入{“ab”，“ba”}这两个字符串中的任意位置。

也就是：P(“abc”) = merge({“cab”，“acb”，“abc”}，{“cba”，“bca”，“bac”})。

最后得出结果：P(“abc”) = {“cab”，“acb”，“abc”，“cba”，“bca”，“bac”}。

既然掌握了其中的套路，我们就可以设计一个递归算法。要生成字符串$s_1...s_n$的所有排列，我们可以先“砍掉”最后一个字符，首先生成$s_1...s_{n-1}$的所有排列。得到$s_1...s_{n-1}$所有排列的结果列表之后，我们会循环遍历这个列表，并在每个字符串的任意位置插入s_n。

简单构造法最后往往会演变成递归法。

方法五：数据结构头脑风暴法

这种方法看起来有点笨，不过很管用。我们可以快速过一遍数据结构的列表，然后逐一尝试各种数据结构。这种方法很实用，因为一旦找到合适的数据结构（比如说树），很多问题也就迎刃而解了。

示例：随机生成一些数字，并保存到一个（可扩展的）数组中。如何跟踪数组的中位数？

6

数据结构头脑风暴法的过程大致如下。

- 链表？恐怕不行——在数字的存取和排序上，链表往往效果不佳。
- 数组？也许可以，不过你已经用了一个数组。你有办法让数组保持有序状态吗？这么做开销恐怕比较大。暂不考虑采用，必要的话，可以回头再试。
- 二叉树？倒也有可能，因为二叉树非常适合处理排序问题。实际上，如果这棵二叉树是完全平衡的，根结点可能就是中位数。不过，你要小心——如果它包含偶数个元素，那么中位数实际上是中间两个元素的平均值。而中间两个元素不可能都是根结点。因此，二叉树也许可行，我们待会再说。
- 堆？堆非常适合基本排序，跟踪最大值和最小值。堆其实也很有意思——只用两个堆，就能跟踪较大的那一半元素和较小的那一半元素。较大的一半保存在小顶堆（min heap）中，其中最小元素位于堆顶。较小的一半则保存在大顶堆（max heap）中，其中最大元素位于堆顶。现在，有了这些数据结构，整个数组的中位数很可能就是两个堆顶之一。如果这两个堆大小不一样，你可以从元素较多的堆中弹出一个元素并压入另一个堆中，两个堆很快就能“重获平衡”。

切记，问题演练得越多，你就越容易判断该选用哪种数据结构。当然了，你也能更自如地从这五种方法中选出最管用的那种。

6.4　怎样才算好代码

至此，你也许明白了，许多公司都想找能写出“优美、整洁”代码的人才。但这到底意味着什么，怎样才能在面试中展现出这方面的能力呢？

一般说来，好代码具备如下特性。

- **正确**：代码应当正确处理所有预期输入（expected input）和非法输入（unexpected input）。
- **高效**：不管是从空间上还是从时间上来衡量，代码都要尽可能地高效运行。所谓的“高效”不仅是指在极限情况下的渐近效率（asymptotic efficiency，大*O*记法），同时也包括实际运行的效率。也就是说，在计算*O*时间时，你可以忽略某个常量因子，但在实际环境中，该常量因子可能有很大影响。
- **简洁**：代码能写成10行就不要写成100行。这样开发人员才能尽快写好代码。
- **易读**：要确保其他开发人员能读懂你的代码，并弄清楚来龙去脉。易读的代码会有适当注释，实现思路也简单易懂。这就意味着，那些包含诸多位操作的花俏的代码不见得就是“好”代码。
- **可维护**：在产品生命周期内，代码经过适当修改就能应对需求的变化。此外，无论对于原开发人员还是其他开发人员，代码都应该易于维护。

力求实现上述特性必须找到一个平衡点。比如，有些情况下，我们往往要牺牲一定的效率好让代码更易维护，有时则要反其道行之。

在面试中，写代码时应该好好考虑这些要素。下文就前面的清单给出更具体的描述。

1. 多用数据结构

假设面试官要求你编写一个函数，对两个简单的多项式求和，其形式为$Ax^a + Bx^b + \cdots$（其中系数和指数为任意正实数或负实数），即多项式的每一项都是一个常量乘以某个数的*n*次幂。面试官还补充说，不必对这些多项式做字符串解析，可以使用任意数据结构来表示它们。

这个函数有多种实现方式。

- 最差的实现方式

最差的实现方式就是将多项式存储为一个double型数组，其中第*k*个元素对应的是多项式中x^k的系数。采用这种结构有一定问题，如此一来，多项式就不能含有负的或非整数指数。要想用这种方法来表示x^{1000}多项式的话，这个数组就得包含1000个元素。

```
int[] sum(double[] poly1, double[] poly2) {
    ...
}
```

- 较差的实现方式

一种不算最差的实现方式是将多项式存为一对数组`coefficients`和`exponents`。采用这种方法，多项式的所有项可以按任意顺序存放，只要系数和指数配对，多项式的第*i*项表示为

coefficients[i] * x$^{exponents[i]}$。

采用这种实现方式，如果coefficients[p] = k和exponents[p] = m，则第p项为kx^m。尽管这么做没有上面那种解法的限制，但还是很凌乱。一个多项式就要用两个数组记录。如果两个数组长度不同，多项式就会出现“未定义”值。而要返回多项式更是麻烦，因为一下子得返回两个数组。

```
??? sum(double[] coeffs1, double[] expon1,
        double[] coeffs2, double[] expon2) {
   ...
}
```

- 较好的实现方式

对于这个问题，较好的实现方式是专为多项式设计一种数据结构。

```
class PolyTerm {
   double coefficient;
   double exponent;
}

PolyTerm[] sum(PolyTerm[] poly1, PolyTerm[] poly) {
   ...
}
```

有些人可能或真的认为这么做“优化过了头”。也许是，也许不是。不管你是不是这么认为，上面的代码都表明你应该用心思考如何设计代码，而不要匆忙地胡乱堆砌一通。

2. 适当重用代码

假设面试官要求你编写一个函数检查某个二进制数（以字符串形式传入）是否等于以字符串表示的十六进制数。

我们可以善用代码重用巧妙解决该问题。

```
public boolean compareBinToHex(String binary, String hex) {
   int n1 = convertToBase(binary, 2);
   int n2 = convertToBase(hex, 16);
   if (n1 < 0 || n2 < 0) {
     return false;
   } else {
     return n1 == n2;
   }
}

public int digitToValue(char c) {
   if (c >= '0' && c <= '9') return c - '0';
   else if (c >= 'A' && c <= 'F') return 10 + c - 'A';
   else if (c >= 'a' && c <= 'f') return 10 + c - 'a';
   return -1;
}

public int convertToBase(String number, int base) {
   if (base < 2 || (base > 10 && base != 16)) return -1;
   int value = 0;
```

```
    for (int i = number.length() - 1; i >= 0; i--) {
      int digit = digitToValue(number.charAt(i));
      if (digit < 0 || digit >= base) {
         return -1;
      }
      int exp = number.length() - 1 - i;
      value += digit * Math.pow(base, exp);
    }
    return value;
}
```

我们本可以实现两套代码，实现二进制数和十六进制数的转换，但这么做只会加大代码的编写难度，而且维护起来也更难。相反，我们还是通过编写convertToBase和digitToValue的方法来重用代码。

3. 模块化

编写模块化代码是指将孤立的代码块划分为相应的方法（函数）。这有助于让代码更易读，可读性和可测试性更强。

假设你在编写交换整数数组中的最大和最小元素的代码，不妨将全部代码写在一个函数里，如下所示：

```
public void swapMinMax(int[] array) {
    int minIndex = 0;
    for (int i = 1; i < array.length; i++) {
      if (array[i] < array[minIndex]) {
         minIndex = i;
      }
    }

    int maxIndex = 0;
    for (int i = 1; i < array.length; i++) {
      if (array[i] > array[maxIndex]) {
         maxIndex = i;
      }
    }

    int temp = array[minIndex];
    array[minIndex] = array[maxIndex];
    array[maxIndex] = temp;
}
```

或者，你还可以采取更模块化的方式，将相对孤立的代码块隔离到对应的方法中。

```
public static int getMinIndex(int[] array) {
    int minIndex = 0;
    for (int i = 1; i < array.length; i++) {
      if (array[i] < array[minIndex]) {
         minIndex = i;
      }
    }
    return minIndex;
}
```

```java

public static int getMaxIndex(int[] array) {
  int maxIndex = 0;
  for (int i = 1; i < array.length; i++) {
    if (array[i] > array[maxIndex]) {
      maxIndex = i;
    }
  }
  return maxIndex;
}

public static void swap(int[] array, int m, int n) {
  int temp = array[m];
  array[m] = array[n];
  array[n] = temp;
}

public static void swapMinMaxBetter(int[] array) {
  int minIndex = getMinIndex(array);
  int maxIndex = getMaxIndex(array);
  swap(array, minIndex, maxIndex);
}
```

虽然前面的非模块化代码看起来也不怎么糟，但模块化代码的一大好处在于它易于测试，因为每一部分都可以单独验证。随着代码越来越复杂，编写模块化代码就变得越发重要。模块化的代码也更易阅读和维护。面试官希望看到你能在面试中展现这些技能。

4. 灵活、健壮

不要因为面试官要求编写代码检查谁是三连棋游戏的赢家，就非得假定它是一个3×3的棋盘。何不放手针对*N*×*N*棋盘编写代码呢？

编写灵活、通用的代码，也可能意味着使用变量，而不是在代码里直接把值写死，或者使用模板/泛型来解决问题。要是有办法编写代码解决更普遍的问题，那我们就应该这么做。

当然，它也有限制条件。如果通用解决方案更为复杂，并且在面试中几乎没有必要使用，那就按照要求解决相应的问题，效果可能会更好。

5. 错误检查

写代码很细心的人有一个明显的特征，那就是她不会想当然地处理输入信息。相反，她会用ASSERT语句或if语句仔细验证输入数据是否合理。

比如，回到前面那段将基数为*i*的进制数（比如基数为2或16）转换成整数的代码。

```java
public int convertToBase(String number, int base) {
  if (base < 2 || (base > 10 && base != 16)) return -1;
  int value = 0;
  for (int i = number.length() - 1; i >= 0; i--) {
    int digit = digitToValue(number.charAt(i));
    if (digit < 0 || digit >= base) {
      return -1;
    }
    int exp = number.length() - 1 - i;
    value += digit * Math.pow(base, exp);
```

```
11     }
12     return value;
13 }
```

在第2行，我们检查基数是否有效（假定除16外，大于10的基数都是无效的，没有标准的字符串表示形式）。在第6行，我们另加了一处错误检查：确保每个数字都落在允许范围内。

诸如此类的错误检查在实际的产品代码中至关重要，因此，面试中也不能掉以轻心。

当然，这些错误检查有时很繁琐，可能会浪费宝贵的面试时间。关键在于指出你会加上错误检查。如果错误检查远非一条if语句就能搞定，写代码时最好先为错误检查预留一些空间，并告诉面试官，完成其余代码之后你会补上错误检查代码。

第 7 章

录用通知及其他

- ❑ 如何处理录用和被拒的情况
- ❑ 如何评估录用待遇
- ❑ 录用谈判
- ❑ 入职须知

7.1　如何处理录用与被拒的情况

面试结束后，刚觉得可以松口气了，你可能又会陷入“面试后综合征”：要接受这家公司的录用吗？它是理想之选吗？如何拒绝录用通知？怎么处置回复期限？我们先来探讨这些问题，接下来几节会细说如何评估录用待遇，以及该怎样讨价还价。

1. 回复期限与延长期限

录用通知大都附有回复期限，一般为一到四周。不过，要是还在苦等其他公司的回音，你可以请求发出录用通知的公司延长回复期限。条件允许的话，大部分公司都会通情达理，予以配合。

2. 如何拒绝录用通知

拒绝公司的录用通知很讲究技巧。即使你现在对该公司不感兴趣，没准几年后又感兴趣了。又或者，该公司与你打过交道的联系人跳到另一家更令人心动的公司。因此，你最好还是礼貌得体地拒绝录用通知，并与该公司做好沟通。

拒绝录用通知时，请给出一个合乎情理且不容置疑的理由。比如，若要舍大公司而取创业公司，你可以阐明自认为创业公司是当下最佳选择的理由。这两种公司截然不同，大公司也不可能突然变成创业公司，所以大公司对此也无可厚非。

3. 处理被拒的情况

科技巨头公司一般会拒掉大约80%的求职者，但他们也明白，这些面试未必能充分考察求职者的能力。有鉴于此，他们通常会给之前被拒的求职者再次面试的机会。甚至有些公司会主动联系以前的求职者，或是因为求职者的面试表现而加快处理流程。

当你接到拒电时，把它视作一时的挫折而非终身裁决。礼貌地感谢招聘人员为此付出的时间和精力，表达自己的遗憾之情，对他们的决定表示理解，并询问什么时候可以再次申请。

找出被拒的原因很难，招聘人员一般也不会吐露实情。当然，如果你拐弯抹角地询问下次面

试该注意哪些事项，运气好的话，说不定可以打探到其中的缘由。你也可以回想一下自己在面试中的表现，但根据我的经验，求职者一般无法作出准确分析。你可能认为是因为自己解决某个问题时大费周折，不过这都是相对的；你并不清楚自己的解题节奏比其他求职者是快还是慢？实际上，一般来说，求职者被拒主要是因为编程与算法功底不过关，总之你要在这些方面狠下工夫。

7.2 如何评估录用待遇

恭喜你！拿到录用通知了！幸运的话，你可能手握不止一个录用通知。现在，招聘人员的工作就是尽其所能说服你签约。那么，又该怎么判断这家公司是否适合自己呢？下面我们将逐一探讨评估录用待遇的若干注意事项。

1. 薪酬待遇的考量

在评估录用通知时，求职者可能会犯的最大错误也许就是过于看重薪水。如此一叶障目导致有些求职者最后反而接受了一个更差的录用通知。薪水只是薪酬待遇的一部分。你还应考虑以下几点。

- **签约奖金、搬家费及其他一次性津贴**：很多公司都会提供签约奖金，有的还会给搬家费。在比较待遇时，最好将这些一次性津贴除以3（或者你预期服务的年限）。
- **各地生活成本差异**：收到多个来自不同地区的录用通知，不要小看地域差别带来的影响。比如，硅谷的生活成本就比西雅图要高出约20%至30%，其中部分原因是加州要收10%的州税，华盛顿州则不用。你可以找几个相关网站来估算各地的生活成本。
- **年终奖**：科技公司的年终奖大约在3%到30%之间。招聘人员可能会告知年终奖的平均数，没有的话，不妨找公司里的朋友打听。
- **股票期权与补助金**：这部分收入也可能是全年收入的另一大块。就像签约奖金一样，你也可以将这部分收入除以3，然后把该数目计入年薪。

当然，切记一点，能学到的知识及公司对你职业生涯的影响远比薪水来得重要。务请慎重考虑当下薪资对你到底有多重要。

2. 职业发展

尽管收到录用通知是如此令人兴奋，甚至有时候幸福感还能持续上几年，但同时你应该开始考虑未来的职业发展方向。因此，现在就思考这份工作会对你的职业发展有怎样的影响，非常重要。也就是，要关注下列问题：

- 该公司名号能否增加自身履历的份量？
- 我能学到多少知识？我会学到相关领域的技术吗？
- 该职位有无升迁可能？开发人员的职业路径是什么样的？
- 想转到管理岗位的话，该公司是否提供了切实可行的通道？
- 该公司或团队是否处于上升期？
- 想要跳槽的话，该公司所在地是否有很多其他机会？我需要搬家吗？

最后一点非常重要，也很容易被人忽视。如果你在微软硅谷分部工作，跳槽时会有许多机

会。然而，要是在微软西雅图总部，选择余地只剩下亚马逊、谷歌和其他一些小公司。此外，要是去了弗吉尼亚州杜勒斯的AOL，那选择余地就更小。所以，千万不要忽视地理位置这个因素，否则你可能被迫在某家公司“终老”，只因为那里没别的公司可去，除非完全改变自己的生活方式。

3. 公司稳定与否

每个人的境遇都有所不同，不过，我一般都会鼓励求职者不要太在意公司稳定与否。真要碰上了裁员，那你肯定也能在同类公司找到一方新天地。你要确认的问题是：要是被解雇了，你会怎么办？你对找到新工作是否信心满满？

4. 幸福指数

当然，幸福指数也是一个重要考量。以下因素都会影响你工作的幸福感。

- **产品**：很多人都非常看重自己做的产品，当然这也是一个重要方面。然而，对大多数工程师来说，还有比这更重要的因素，比如，与哪些人一起共事。
- **经理与队友**：当人们提及自己热爱或痛恨自己的工作时，通常是他们的队友与经理占了主因。你有没有跟未来的经理、队友碰过面？你喜欢和他们交流吗？
- **企业文化**：企业文化涉及方方面面，从如何作决策到整体氛围及公司的组织架构。不妨问问未来的同事，看看他们会如何描述公司的企业文化。
- **工作时长**：问一问未来的队友，他们一般工作多长时间，确定是否契合自己的生活节奏。不过，值得注意的是，临近产品发布时，加班在所难免。

此外，你还要看看是否有机会在不同的团队轮岗（比如在谷歌就很宽松），万一不喜欢，你还有机会找到更合适的团队和部门。

7.3 录用谈判

2010年年末，我报了一个谈判训练班。第一天，培训师让我们设想一个购车的场景。经销商A报的是一口价，2万美元。而经销商B允许议价。那么，要讲下多少钱你才愿意去经销商B那里买车呢？（快！迅速报出你的答案！）

最后，全班给出的平均数目是便宜750美元。换言之，学员们都愿意付750美元，免除一小时的讨价还价。这也没什么奇怪的，在对全班学员进行的民调中，大部分人都表示自己接受工作录用时也不会讨价还价。公司给多少就是多少。

拜托，请理直气壮地还还价吧。下面是几点可资参考的建议。

(1) **要理直气壮**。是的，迈出第一步很难，没什么人喜欢谈判。但讨价还价还是有必要的。招聘人员不会因为你有异议就撤回录用通知，所以你也不会有什么损失。

(2) **最好手头有其他选择**。从根本上来说，招聘人员愿意与你谈判是因为他们希望你能加入公司。如果你手头有其他选择，他们就会更担心你有可能拒绝他们的录用邀约。

(3) **提出具体的“要价”**。给一个具体的数目，比如要求年薪增加7千美金会比泛泛地要求涨薪效果更佳。毕竟，如果只是要求涨薪，招聘人员可以不痛不痒地加个1千块来打发你。

(4) **开出比预期稍高的价码**。在谈判中，人们一般不会全盘接受你的要求，总是要讨价还价一番。因此，你开的价码可以比自己预期的高一些，这样公司再往下降一降，最后皆大欢喜。

(5) **不要只盯着薪水**。公司更愿意就薪水之外的条件作出让步，因为给你大幅涨薪可能会造成团队内部同工不同酬的情况。你可以稍作变通，要求更多的期权或签约奖金。同样，还可以要求公司将搬家费直接折算成现金。这对应届毕业生来说更划算，因为他们家什少，搬家也花不了多少钱。

(6) **使用最合适的方法**。很多人会建议你通过电话进行谈判。在一定程度上，他们是对的。当然，要是不喜欢在电话中讨价还价，可以使用电子邮件。最重要的是你本人有谈判的想法，效果比形式更重要。

此外，与大公司进行谈判，你要了解这些公司都有某种职等级别制度，一定的级别对应一定的薪资范围。微软对此就有明确的规定。你可以在对应范围内讨价还价，但要价太高就会超出这个范围。如果你觉得自己可以拿到更高级别，那就得向招聘人员和未来的团队证明你有这个实力——谈判过程会比较难，但也不是没有可能。

7.4　入职须知

入职不是终点，而是你职业生涯的新起点。一旦正式加入一家公司，你就得开始做好职业规划。你想达到什么样的目标，如何才能实现？

1. 制定时间表

“入此门后，非疯即癫”，这种情况很常见。新生活开始之际总是很美好的。可五年之后，你还停留在原地不动。到那时才意识到自己虚度了最近三年的时光，技术没什么长进，履历也乏善可陈。当初为什么不待上两年就走呢？

志得意满之际反而是最危险的时候，让你“温水煮青蛙”而忘记了百尺竿头更进一步。这也正是工作伊始就要做好职业规划的原因。好好想一想，十年后想干什么？该如何一步步达成目标？此外，每年都要总结一下过去一年自己在职业与技能上取得了哪些进步，明年又有什么样的规划？

提前做好规划并定期对照检查，这样，就能避免自己陷入“温水煮青蛙”的困境。

2. 打造坚实的人际网络

在找新工作时，人际网络的作用很大。毕竟，在线申请工作有很多不确定因素；有人推荐的话就会好很多，而这取决于你的关系网有多强大。

所以，在工作中要与经理、同事建立良好的关系。就算有人离职，你们也可以继续保持联系。比如，在他们离职几周后，写封简短的邮件问候一下，这不仅可以拉近你们的距离，还可以将原本的同事关系升华为朋友关系。

这些小技巧同样适用于你的个人生活。你的朋友、朋友的朋友都是你的宝贵资源。我为人人，人人为我。

3. 向经理寻求帮助

有些经理很愿意提携下属，帮助开拓职业道路，但也有些人会不闻不问。所以，这都要看你自己是否有心开拓进取，寻求更好的职业发展。

请开诚布公地向你的主管表明心迹。如欲从事更多后端编程项目，不妨直言相告。如要往管理层发展，你可以与经理探讨自己需要做些什么。

记得时时为自己打气，这样才能逐步实现既定目标。

第 8 章

面试考题

请登录我们的网站www.CrackingTheCodingInterview.com，下载完整可编译的Java/Eclipse工程，并与其他读者一起讨论书中的面试题，提交问题，查看本书勘误，发布简历及寻求其他建议。

数据结构

- 数组与字符串
- 链表
- 栈与队列
- 树与图

概念与算法

- 位操作
- 智力题
- 数学与概率
- 面向对象设计
- 递归和动态规划
- 扩展性与存储限制
- 排序与查找
- 测试

知识类问题

- C和C++
- Java
- 数据库
- 线程与锁

附加面试题

- 中等难题
- 高难度题

8.1 数组与字符串

想必本书读者都很熟悉什么是数组和字符串，因此这里不再赘述细节。我们会把重心放在这些数据结构相关的一些常见技巧和问题上。

请注意，数组问题与字符串问题往往是相通的。换句话说，书中提到的数组问题也可能以字符串的形式出现，反之亦然。

1. 散列表

散列表是一种将键（key）映射为值（value）从而实现快速查找的数据结构。在简易实现中，散列表包含一个底层数组和一个*散列函数*（hash function）。插入一个对象及对应的键时，散列函数会将键映射为数组的一个索引。然后，这个对象就会储存到数组中该索引对应的位置。

不过，通常情况下，这个方法还不够完善。在上面的实现中，所有可能的键必须转化为各不相同的散列值，否则一不小心就可能改写某些数据。因此，为了防止这类“碰撞冲突”，这个数组会变得非常大，以便放下所有可能的键。

除了创建按索引`hash(key)`储存对象的超大数组，我们还可以选用小得多的数组，并将对象储存在索引为`hash(key) % array_length`的数组元素指向的链表中。要通过某个键来查找对象，就必须根据散列值找到对应的链表，然后在链表中查找相应的键。

另外，我们还可以采用二叉查找树来实现散列表。只要我们让这棵树保持平衡，就能保证数据查找用时为$O(\log n)$。此外，这种实现占用的空间可能更少，原因很简单，我们不必一开始就分配一个大数组。

面试之前，建议你多加练习，掌握散列表的实现和用法。散列表是面试中最常见的数据结构之一，相关问题也是技术面试的常客。

下面是一段使用散列表的简单Java程序。

```java
public HashMap<Integer, Student> buildMap(Student[] students) {
   HashMap<Integer, Student> map = new HashMap<Integer, Student>();
   for (Student s : students) map.put(s.getId(), s);
   return map;
}
```

注意，尽管有时面试官会明确要求使用散列表，但多半还是要靠你自己想到用散列表解决问题。

2. ArrayList（动态数组）

ArrayList，即动态数组，是一种按需动态调整大小的数组，数据访问时间为$O(1)$。一种典型的实现是在数组存满时将其扩容两倍。每次扩容用时$O(n)$，不过这种操作频次极少，因此均摊下来访问时间仍为$O(1)$。

```java
public ArrayList<String> merge(String[] words, String[] more) {
   ArrayList<String> sentence = new ArrayList<String>();
   for (String w : words) sentence.add(w);
   for (String w : more) sentence.add(w);
   return sentence;
}
```

8

3. StringBuffer

假设你要将一组字符串拼接起来，如下所示。这段代码会运行多长时间？为简单起见，假设所有字符串等长（皆为x），一共有n个字符串。

```
public String joinWords(String[] words) {
   String sentence = "";
   for (String w : words) {
     sentence = sentence + w;
   }
   return sentence;
}
```

每次拼接都会新建一个字符串，包含原有两个字符串的全部字符。第一次循环要拷贝x个字符，第二次循环要拷贝$2x$个字符，第三次要拷$3x$个，依此类推。综上，这段代码的时间开销为$O(x + 2x + \cdots + nx)$，可简化为$O(xn^2)$。为什么不是$O(xn^n)$？因为$1 + 2 + \cdots + n$等于$n(n+1)/2$，即$O(n^2)$。

`StringBuffer`可以避免上面的问题。它会直接创建一个足以容纳所有字符串的数组，等到拼接完成才将这些字符串转成一个字符串。

```
public String joinWords(String[] words) {
   StringBuffer sentence = new StringBuffer();
   for (String w : words) {
     sentence.append(w);
   }
   return sentence.toString();
}
```

不妨试着自己实现一把`StringBuffer`，这对你掌握字符串、数组和常见数据结构大有裨益。

面试题目

1.1 实现一个算法，确定一个字符串的所有字符是否全都不同。假使不允许使用额外的数据结构，又该如何处理？（第108页）

1.2 用C或C++实现`void reverse(char* str)`函数，即反转一个null结尾的字符串。（第109页）

1.3 给定两个字符串，请编写程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。（第109页）

1.4 编写一个方法，将字符串中的空格全部替换为“%20”。假定该字符串尾部有足够的空间存放新增字符，并且知道字符串的“真实”长度。（注：用Java实现的话，请使用字符数组实现，以便直接在数组上操作。）（第111页）

示例

输入：`"Mr John Smith "`

输出：`"Mr%20John%20Smith"`

1.5 利用字符重复出现的次数，编写一个方法，实现基本的字符串压缩功能。比如，字符串“aabcccccaaa”会变为“a2b1c5a3”。若“压缩”后的字符串没有变短，则返回原先的字符串。（第112页）

1.6 给定一幅由$N \times N$矩阵表示的图像，其中每个像素的大小为4字节，编写一个方法，将图像旋转90度。不占用额外内存空间能否做到？（第114页）

1.7 编写一个算法，若$M \times N$矩阵中某个元素为0，则将其所在的行与列清零。（第115页）

1.8 假定有一个方法`isSubstring`，可检查一个单词是否为其他字符串的子串。给定两个字符串`s1`和`s2`，请编写代码检查`s2`是否为`s1`旋转而成，要求只能调用一次`isSubstring`。（比如，`waterbottle`是`erbottlewat`旋转后的字符串。）（第116页）

参考问题：位操作（#5.7）；面向对象设计（#8.10）；递归（#9.3）；排序与查找（#11.6）；C++（#13.10）；中等难题（#17.7、#17.8、#17.14）。

8.2　链表

链表问题有时会难倒不少求职者，因为链表元素访问用时不定，而且往往涉及递归。不过，好在链表问题不是变化多端，许多问题只是在常见问题的基础上稍作调整而已。

链表问题非常依赖基本概念，对于求职者来说，可以从无到有实现链表是一项基本要求。下面是我们给出的参考实现代码。

1. 创建链表

下面的代码实现了一个非常基本的单向链表。

```
class Node {
   Node next = null;
   int data;

   public Node(int d) {
     data = d;
   }

   void appendToTail(int d) {
     Node end = new Node(d);
     Node n = this;
     while (n.next != null) {
        n = n.next;
     }
     n.next = end;
   }
}
```

切记，在面试中遇到链表题时，务必弄清楚它到底是单向链表还是双向链表。

2. 删除单向链表中的结点

删除单向链表中的结点非常简单。给定一个结点`n`，我们先找到它的前趋结点`prev`，并将`prev.next`设置为`n.next`。如果这是双向链表，我们还要更新`n.next`，将`n.next.prev`置为`n.prev`。当然，我们必须注意：(1) 检查空指针；(2) 必要时更新表头（head）或表尾（tail）指针。

此外，如果采用C、C++或其他要求开发人员自行管理内存的语言，还应考虑要不要释放删除结点的内存。

```
Node deleteNode(Node head, int d) {
   Node n = head;

   if (n.data == d) {
     return head.next; /* 表头指向下一结点 */
   }

   while (n.next != null) {
     if (n.next.data == d) {
        n.next = n.next.next;
        return head; /* 表头不变 */
     }
     n = n.next;
   }
   return head;
}
```

3. “快行指针”技巧

在处理链表问题时，“快行指针”（runner，或称第二个指针）是一种很常见的技巧。“快行指针”指的是同时用两个指针来迭代访问链表，只不过其中一个比另一个超前一些。“快”指针往往先行几步，或与“慢”指针相差固定的步数。

举个例子，假定有一个链表a_1->a_2->…->a_n->b_1->b_2->…->b_n，你想将其重新排列成a_1->b_1->a_2->b_2->…->a_n->b_n 。另外，你不知道该链表的长度（但确定它有偶数个元素）。

你可以用两个指针，其中p1（快指针）每次都向前移动两步，而同时p2只移动一步。当p1到达链表末尾时，p2刚好位于链表中间位置。然后，再让p1与p2一步步从尾向头反向移动，并将p2指向的结点插入到p1所指结点后面。

4. 递归问题

许多链表问题都要用到递归。解决链表问题碰壁时，不妨试试递归法能否奏效。这里暂时不会深入探讨递归，后面会有专门章节予以讲解。

当然，还需注意递归算法至少要占用$O(n)$空间，其中n为递归调用的层数。实际上，所有递归算法都可以转换成迭代法，只是后者实现起来可能要复杂得多。

面试题目

2.1 编写代码，移除未排序链表中的重复结点。（第117页）

进阶

如果不得使用临时缓冲区，该怎么解决？

2.2 实现一个算法，找出单向链表中倒数第k个结点。（第118页）

2.3 实现一个算法，删除单向链表中间的某个结点，假定你只能访问该结点。（第120页）

示例

输入：单向链表a->b->c->d->e中的结点c。

结果：不返回任何数据，但该链表变为：a->b->d->e

2.4 编写代码，以给定值x为基准将链表分割成两部分，所有小于x的结点排在大于或等于x的结点之前。（第121页）

2.5 给定两个用链表表示的整数，每个结点包含一个数位。这些数位是反向存放的，也就是个位排在链表首部。编写函数对这两个整数求和，并用链表形式返回结果。（第123页）

示例

输入：`(7-> 1 -> 6) + (5 -> 9 -> 2)`，即`617 + 295`。

输出：`2 -> 1 -> 9`，即`912`。

进阶

假设这些数位是正向存放的，请再做一遍。

示例

输入：`(6 -> 1 -> 7) + (2 -> 9 -> 5)`，即`617 + 295`。

输出：`9 -> 1 -> 2`，即`912`。

2.6 给定一个有环链表，实现一个算法返回环路的开头结点。（第126页）

有环链表的定义

在链表中某个结点的`next`元素指向在它前面出现过的结点，则表明该链表存在环路。

示例

输入：`A -> B -> C -> D -> E -> C`（C结点出现了两次）。

输出：`C`

2.7 编写一个函数，检查链表是否为回文。（第128页）

参考问题：树与图（#4.4）；面向对象设计（#8.10）；扩展性与内存限制（#10.7）；中等难题（#17.13）。

8.3　栈与队列

和链表问题一样，熟练掌握数据结构的基本原理，栈与队列问题处理起来要容易得多。当然，有些问题也可能相当棘手。部分问题不过是对基本数据结构略作调整，而其他问题则要难得多。

1. 实现一个栈

栈采用后进先出（LIFO）顺序。换言之，像一堆盘子那样，最后入栈的元素最先出栈。

下面给出了栈的简单实现代码。注意，栈也可以用链表实现。实际上，栈和链表本质上是一样的，只不过用户通常只能看到栈顶元素。

```
class Stack {
   Node top;

   Object pop() {
     if (top != null) {
       Object item = top.data;
       top = top.next;
```

8

```
      return item;
    }
    return null;
  }

  void push(Object item) {
    Node t = new Node(item);
    t.next = top;
    top = t;
  }

  Object peek() {
    return top.data;
  }
}
```

2. 实现一个队列

队列采用先进先出（FIFO）顺序。就像一支排队购票的队伍那样，最早入列的元素也是最先出列的。

队列也可以用链表实现，新增元素追加至表尾。

```
class Queue {
  Node first, last;

  void enqueue(Object item) {
    if (first == null) {
      last = new Node(item);
      first = last;
    } else {
      last.next = new Node(item);
      last = last.next;
    }
  }

  Object dequeue() {
    if (first != null) {
      Object item = first.data;
      first = first.next;
      return item;
    }
    return null;
  }
}
```

面试题目

3.1 描述如何只用一个数组来实现三个栈。（第131页）

3.2 请设计一个栈，除pop与push方法，还支持min方法，可返回栈元素中的最小值。push、pop和min三个方法的时间复杂度必须为$O(1)$。（第135页）

3.3 设想有一堆盘子，堆太高可能会倒下来。因此，在现实生活中，盘子堆到一定高度时，我们就会另外堆一堆盘子。请实现数据结构`SetOfStacks`，模拟这种行为。`SetOfStacks`应该由多个栈组成，并且在前一个栈填满时新建一个栈。此外，`SetOfStacks.push()`和`SetOfStacks.pop()`应该与普通栈的操作方法相同（也就是说，`pop()`返回的值，应该跟只有一个栈时的情况一样）。（第137页）

进阶

实现一个`popAt(int index)`方法，根据指定的子栈，执行pop操作。

3.4 在经典问题汉诺塔中，有3根柱子及*N*个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自底向上从大到小依次套在第一根柱子上（即每一个盘子只能放在更大的盘子上面）。移动圆盘时有以下限制：

(1) 每次只能移动一个盘子；

(2) 盘子只能从柱子顶端滑出移到下一根柱子；

(3) 盘子只能叠在比它大的盘子上。

请运用栈，编写程序将所有盘子从第一根柱子移到最后一根柱子。（第140页）

3.5 实现一个`MyQueue`类，该类用两个栈来实现一个队列。（第142页）

3.6 编写程序，按升序对栈进行排序（即最大元素位于栈顶）。最多只能使用一个额外的栈存放临时数据，但不得将元素复制到别的数据结构中（如数组）。该栈支持如下操作：`push`、`pop`、`peek`和`isEmpty`。（第143页）

3.7 有家动物收容所只收容狗与猫，且严格遵守“先进先出”的原则。在收养该收容所的动物时，收养人只能收养所有动物中“最老”（根据进入收容所的时间长短）的动物，或者，可以挑选猫或狗（同时必须收养此类动物中“最老”的）。换言之，收养人不能自由挑选想收养的对象。请创建适用于这个系统的数据结构，实现各种操作方法，比如`enqueue`、`dequeueAny`、`dequeueDog`和`dequeueCat`等。允许使用Java内置的`LinkedList`数据结构。（第145页）

参考问题：链表（#2.7）；数学与概率（#7.7）。

8.4 树与图

许多求职者会觉得树与图的问题是最难对付的。检索这两种数据结构比数组或链表等线性数据结构要复杂得多。此外，在最坏情况和平均情况下，检索用时可能差别巨大，对于任意算法，我们都要从这两方面进行评估。能够自如地从无到有实现树或图对求职者而言非常重要。

1. 需要注意的潜在问题

树与图的问题容易出现含糊的细节和错误的假设。务必留意下列问题，必要时寻求澄清。

- 二叉树与二叉查找树

碰到二叉树问题时，许多求职者会假定面试官问的是二叉查找树。此时务必问清楚二叉树是否为二叉查找树。二叉查找树附加有如下条件：对于任意结点，左子结点小于或等于当前结点，后者又小于所有右子结点。

8

- 平衡与不平衡

许多树都是平衡的，但并非全都如此。树是否平衡要找面试官确认。如果树是不平衡的，你应当从平均情况和最坏情况所需时间来描述自己的算法。注意，树的平衡有多种方法，平衡一棵树只意味着子树的深度差不会超过一定值，并不表示左子树和右子树的深度完全相同。

- 完满和完整（Full and Complete）

完满和完整树的所有叶结点都在树的底部，所有非叶结点都有两个子结点。注意完满和完整树极其稀少，因为一棵树必须正好有2^n-1个结点才能满足这个条件。

2. 二叉树遍历

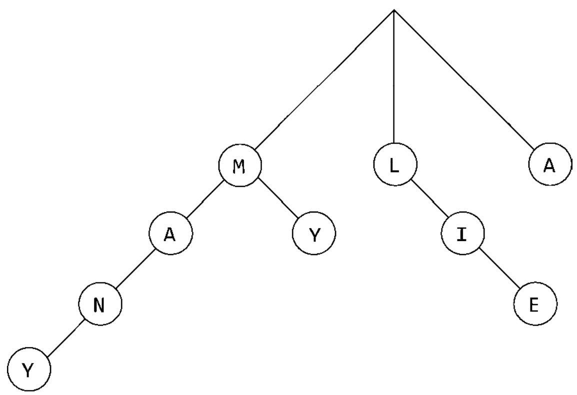
面试之前，你应该能够熟练实现中序、后序和前序遍历。其中最常见的是中序遍历，先遍历左子树，然后访问当前结点，最后遍历右子树。

3. 树的平衡：红黑树和平衡二叉树

学习如何实现平衡树可助你成为更好的软件工程师，只不过面试中很少会问及平衡树。你应该熟悉平衡树各种操作的执行时间，大致了解如何平衡一棵树。当然，就面试而言，掌握个中细节也许没什么必要。

4. 单词查找树（trie）

trie树是*n*层树的一种变体，其中每个结点存储有字符。整棵树的每条路径自上而下表示一个单词。一棵简单的trie树类似下图：



5. 图的遍历

大部分求职者都比较熟悉二叉树的遍历，但图的遍历则要难得多。广度优先搜索（BFS）更是难上加难。

值得注意的是，广度优先搜索（BFS）和深度优先搜索（DFS）通常用于不同的场景。如要访问图中所有结点[①]，或者访问最少的结点直至找到想找的结点，DFS一般最为简单。不过，如果一棵树的规模非常大，在离最初结点太远时想要随时退出的话，DFS可能会有问题；我们可能搜索了该结点的成千上万个祖先结点，却还未搜索该结点的全部子结点。对于这些情况，一般首选BFS。

① 图中的“结点”一般称为顶点，这里依原文译作结点。——译者注

● 深度优先搜索（DFS）

在DFS中，我们会访问结点r，然后循环访问r的每个相邻结点。在访问r的相邻结点n时，我们会在继续访问r的其他相邻结点之前先访问n的所有相邻结点。也就是说，在继续搜索r的其他子结点之前，我们会先穷尽搜索n的子结点。

注意，前序和树遍历的其他形式都是一种DFS。主要区别在于，对图实现该算法时，我们必须先检查该结点是否已访问。如果不这么做，就可能陷入无限循环。

下面是实现DFS的伪代码。

```
void search(Node root) {
   if (root == null) return;
   visit(root);
   root.visited = true;
   foreach (Node n in root.adjacent) {
     if (n.visited == false) {
        search(n);
     }
   }
}
```

● 广度优先搜索（BFS）

BFS相对不太直观，除非之前熟悉BFS的实现，否则大部分求职者都会觉得它很难对付。

在BFS中，我们会在搜索r的“孙子结点”之前先访问结点r的所有相邻结点。用队列实现的迭代方案往往最有效。

```
void search(Node root) {
   Queue queue = new Queue();
   root.visited = true;
   visit(root);
   queue.enqueue(root); // 加至队列尾部

   while (!queue.isEmpty()) {
     Node r = queue.dequeue(); // 从队列头部移除
     foreach (Node n in r.adjacent) {
        if (n.visited == false) {
           visit(n);
           n.visited = true;
           queue.enqueue(n);
        }
     }
   }
}
```

当面试官要求你实现BFS时，切记关键在于队列的使用。用了队列，这个算法的其余部分自然也就成型了。

面试题目

4.1 实现一个函数，检查二叉树是否平衡。在这个问题中，平衡树的定义如下：任意一个结点，其两棵子树的高度差不超过1。（第146页）

4.2 给定有向图，设计一个算法，找出两个结点之间是否存在一条路径。（第148页）

4.3 给定一个有序整数数组，元素各不相同且按升序排列，编写一个算法，创建一棵高度最小的二叉查找树。（第149页）

4.4 给定一棵二叉树，设计一个算法，创建含有某一深度上所有结点的链表（比如，若一棵树的深度为D，则会创建出D个链表）。（第150页）

4.5 实现一个函数，检查一棵二叉树是否为二叉查找树。（第151页）

4.6 设计一个算法，找出二叉查找树中指定结点的“下一个”结点（也即中序后继）。可以假定每个结点都含有指向父结点的连接。（第154页）

4.7 设计并实现一个算法，找出二叉树中某两个结点的第一个共同祖先。不得将额外的结点储存在另外的数据结构中。注意：这不一定是二叉查找树。（第155页）

4.8 你有两棵非常大的二叉树：T1，有几百万个结点；T2，有几百个结点。设计一个算法，判断T2是否为T1的子树。

如果T1有这么一个结点n，其子树与T2一模一样，则T2为T1的子树。也就是说，从结点n处把树砍断，得到的树与T2完全相同。（第159页）

4.9 给定一棵二叉树，其中每个结点都含有一个数值。设计一个算法，打印结点数值总和等于某个给定值的所有路径。注意，路径不一定非得从二叉树的根结点或叶结点开始或结束。（第161页）

参考问题：扩展性与存储限制（#10.2、#10.5）；排序与查找（#11.8）；中等难题（#17.13、#17.14）；高难度题（#18.6、#18.8、#18.9、#18.10、#18.13）。

8.5　位操作

位操作可以用于解决各种各样的问题。有时候，有的问题会明确要求用位操作来解决，而在其他情况下，位操作也是优化代码的实用技巧。写代码要熟悉位操作，同时也要熟练掌握位操作的手工运算。处理位操作问题时，务必小心翼翼，不经意间就会犯下各种小错。代码写好后一定要进行充分的测试，也可以边写代码边测试。

1. 手工位操作

如果像很多人一样，你也惧怕位操作问题，以下这些练习对你大有裨益。当你一筹莫展或困惑不解时，不妨换用十进制来理解相关操作，再将这些操作过程应用到二进制上。

记住，符号^表示XOR（异或）操作，~表示非（取反）操作。为简单起见，假定操作数的位宽为4位。我们可以手工或是施以若干技巧（详情如下）解决下表第三列的问题。

0110 + 0010	0011 * 0101	0110 + 0110
0011 + 0010	0011 * 0011	0100 * 0011
0110 - 0011	1101 >> 2	1101 ^ (~1101)
1000 - 0110	1101 ^ 0101	1011 & (~0 << 2)

答案：第一行（1000, 1111, 1100）；第二行（0101, 1001, 1100）；第三行（0011, 0011, 1111）；第四行（0010, 1000, 1000）。

第三列问题的解决技巧如下。

(1) `0110 + 0110`相当于`0110 * 2`，也就是将`0110`左移1位。

(2) `0100`等于4，`0100 * 0011`也就是将`0011`乘以4。一个数与2^n相乘，相当于将这个数左移n位。于是，将`0011`左移2位得到`1100`。

(3) 逐个比特分解这一操作。一个比特与对它取反的值做异或操作，结果总是1。因此，`a^(~a)`的结果是一串1。

(4) 类似`x & (~0 << n)`的操作会将x最右边的n位清零。`~0`的值就是一串1，将它左移n位后的结果为一串1后面跟n个0。将这个数与x进行“位与”操作，相当于将x最右边的n位清零。

要处理其他问题，不妨打开Windows上的计算器，选择“查看”（View）菜单项，再点选该工具的“程序员”（Programmer）版本。有了这个应用程序，就可以执行位与、异或和移位等各种二进制运算。

2. 位操作原理与技巧

处理位操作问题时，理解以下原理会大有帮助。不要一味死记硬背，而应思考这些等式何以成立。在下面的示例中，“1s”和“0s”分别表示一串1和一串0。

```
x ^ 0s = x      x & 0s = 0      x | 0s = x
x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
x ^ x = 0       x & x = x       x | x = x
```

要理解这些表达式的含义，你必须记住所有操作是按位进行的，某一位的运算结果不会影响其余位。也就是说，只要上述语句对某一位成立，则同样适用于一串位。

3. 常见位操作：获取、设置、清除及更新位数据

以下这些位操作非常重要，不过切忌死记硬背，否则只会滋生一些难以察觉的错误。相反，你要吃透这些操作方法，学会举一反三，灵活处理其他问题。

- 获取

该方法将1左移i位，得到形如`00010000`的值。接着，对这个值与num执行“位与”操作，从而将i位之外的所有位清零。最后，检查该结果是否为零。不为零说明i位为1，否则，i位为0。

```
boolean getBit(int num, int i) {
   return ((num & (1 << i)) != 0);
}
```

- 置位

`setBit`先将1左移i位，得到形如`00010000`的值。接着，对这个值和num执行“位或”操作，这样只会改变i位的数据。该掩码i位除外的位均为零，故而不会影响num的其余位。

```
int setBit(int num, int i) {
   return num | (1 << i);
}
```

- 清零

该方法与`setBit`刚好相反。首先，将1左移i位取得形如`00010000`的值，对这个值取反进而得到类似`11101111`的掩码。接着，对该掩码和num执行“位与”操作。这样只会清零num的i位，其

余位则保持不变。

```
int clearBit(int num, int i) {
   int mask = ~(1 << i);
   return num & mask;
}
```

将num最高位至*i*位（含）清零的做法如下：

```
int clearBitsMSBthroughI(int num, int i) {
   int mask = (1 << i) - 1;
   return num & mask;
}
```

将*i*位至0位（含）清零的做法如下：

```
int clearBitsIthrough0(int num, int i) {
   int mask = ~((1 << (i+1)) - 1);
   return num & mask;
}
```

- 更新

这个方法将setBit与clearBit合二为一。首先，用诸如11101111的掩码将num的第*i*位清零。接着，将待写入值*v*左移*i*位，得到一个*i*位为*v*但其余位都为0的数。最后，对之前取得的两个结果执行“位或”操作，*v*为1则将num的*i*位更新为1，否则该位仍为0。

```
int updateBit(int num, int i, int v) {
   int mask = ~(1 << i);
   return (num & mask) | (v << i);
}
```

面试题目

5.1　给定两个32位的整数*N*与*M*，以及表示比特位置的*i*与*j*。编写一个方法，将*M*插入*N*，使得*M*从*N*的第*j*位开始，到第*i*位结束。假定从*j*位到*i*位足以容纳*M*，也即若*M*=10011，那么*j*和*i*之间至少可容纳5个位。例如，不可能出现*j* = 3和*i* = 2的情况，因为第3位和第2位之间放不下*M*。（第163页）

示例

输入：N = 10000000000, M = 10011, i = 2, j = 6

输出：N = 10001001100

5.2　给定一个介于0和1之间的实数（如0.72），类型为double，打印它的二进制表示。如果该数字无法精确地用32位以内的二进制表示，则打印“ERROR”。（第164页）

5.3　给定一个正整数，找出与其二进制表示中1的个数相同、且大小最接近的那两个数（一个略大，一个略小）。（第165页）

5.4　解释代码((n & (n-1)) == 0)的具体含义。（第170页）

5.5　编写一个函数，确定需要改变几个位，才能将整数*A*转成整数*B*。（第171页）

示例

输入：`31, 14`
输出：`2`

5.6 编写程序，交换某个整数的奇数位和偶数位，使用指令越少越好（也就是说，位0与位1交换，位2与位3交换，依此类推）。（第171页）

5.7 数组A包含0到n的所有整数，但其中缺了一个。在这个问题中，只用一次操作无法取得数组A里某个整数的完整内容。此外，数组A的元素皆以二进制表示，唯一可用的访问操作是“从`A[i]`取出第j位数据”，该操作的时间复杂度为常数。请编写代码找出那个缺失的整数。你有办法在$O(n)$时间内完成吗？（第172页）

5.8 有个单色屏幕存储在一个一维字节数组中，使得8个连续像素可以存放在一个字节里。屏幕宽度为w，且w可被8整除（即一个字节不会分布在两行上），屏幕高度可由数组长度及屏幕宽度推算得出。请实现一个函数`drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)`，绘制从点(x_1, y)到点(x_2, y)的水平线。（第174页）

参考问题：数组与字符串（#1.1、#1.7）；递归（#9.4、#9.11）；扩展性与存储限制（#10.3、#10.4）；C++（#13.9）；中等难题（#17.1、#17.4）；高难度题（#18.1）。

8.6 智力题

智力题当属最有争议的面试题之列，很多公司甚至明文规定面试中不得出现智力题。尽管如此，你还是会时不时地碰到它。为什么呢？因为人们对于智力题尚无明确的定义。

不过，好在哪怕你碰到了这类问题，一般来说它们也不会太难。你不需要做脑筋急转弯，并且几乎总有办法通过逻辑推理得出答案。很多智力题甚至还涉及数学或计算机科学的基础知识。

下面，我们会列举一些应对智力题的常见方法。

1. 大声说出你的思路

遇到智力题时，切忌惊慌。就像算法题一样，面试官只不过想看看你会如何处理难题；他们并不期待你立即给出正确答案。只管大声说出解题思路，让面试官了解你的应对之道。

2. 总结规律和模式

很多情况下，你会发现，把解题过程中发现的“规律”或“模式”写下来帮助很大。并且，你确实应该这么做，这有助于加深记忆。下面会举例说明这种方法。

给定两条绳子，每条绳子燃烧殆尽正好要用一个小时。怎样用这两条绳子准确计量15分钟？注意这些绳子密度不均匀，因此烧掉半截绳子不一定正好用半个小时。

技巧：先别急着往下看，不妨试着自己解决此问题。一定要看下面的提示信息的话——也请一段一段慢慢看。后续段落会逐步揭晓答案。

8

从题目可知，计量一小时不成问题。当然也可以计量两小时，先点燃一根绳子，等它燃烧殆尽，再点燃第二根。由此我们总结出第一条规律。

规律1：给定两条绳子，燃烧殆尽各需x分钟和y分钟，我们可以计时$x+y$分钟。

那么，还有其他烧绳子的花样吗？当然，我们知道从中间（或绳子两头以外的任意位置）点燃绳子没什么用。火苗会向绳子两头蔓延，我们不知道过多久才会烧完。

话说回来，我们可以同时点燃绳子两头。30分钟后火焰便会在绳子某个位置汇合。

规律2：给定一条需要x分钟烧完的绳子，我们可以计时$x/2$分钟。

由此可知，用一条绳子可以计时30分钟。这就意味着我们可以在燃烧第二条绳子时减去这30分钟，也就是点燃第一条绳子两头的同时，只点燃第二条绳子的一头。

规律3：烧完绳子1用时x分钟，绳子2用时y分钟，则可以用第二条绳子计时$(y-x)$分钟或$(y-x/2)$分钟。

综合以上规律，不难得出：既然可以用绳子2计时30分钟，再适时点燃绳子2的另一头（见规律2），则15分钟后绳子2便会燃烧殆尽。

将上面的做法从头至尾整理如下。

(1) 点燃绳子1两头的同时，点燃绳子2的一头。

(2) 当绳子1从两头烧至中间某个位置时，正好过去30分钟。而绳子2还可以再烧30分钟。

(3) 此时，点燃绳子2的另一头。

(4) 15分钟后，绳子2将全部烧完。

从中可以看出，只要一步步归纳规律，并在此基础上进行总结，智力题便可迎刃而解。

3. 略作变通

许多智力题往往涉及将最坏情况减至最低限度的问题，措辞上要么要求尽可能减少步骤，要么限定具体的试验次数。一种实用的技巧是尝试“平衡”最坏情况。也就是说，如果早先的解决方案效果不太理想，我们可以针对最坏情况略作变通。用一个例子来解释会更为清晰。

“九球称重”是一个经典面试题。给定9个球，其中8个球的重量相同，只有一个比较重。然后给定一个天平，可以称出左右两边哪边更重。最多用两次天平，找出这个重球。

第一种做法是将球分成2组，4个一组，第9个球暂时搁在一边。如果有一组球较重，则重球必在其中；但如果两组球重量相同，则第9个球为重球。按此思路将包含重球的这一组球再分成两组，在最坏情况下我们需要称量3次——多了一次！

因此，这是一个“失衡”的解法：如果第9个球是重球，我们只需称量一次；但如果不是，则需称量3次。如果我们略作调整，将更多的球与第9个球配在一起，就不会出现“失衡”的状况。这就是所谓“最坏情况下的平衡”。

现在，将这些球均分成3个一组共3组，称量一次就能知道哪一组球更重。我们甚至可以总结出一条规律：给定N个球，其中N能被3整除，称量一次便能找到包含重球的那一组球。

找到这一组3个球之后，我们只是简单地重复此前的模式：先把一个球放到一边，称量剩下的两个球。从中挑出那个重球；或者，如果这两个球重量相同，那第3个球便是重球。

4. 触类旁通

要是卡壳了，不妨考虑运用前面提到的算法题的五种解法。剔除技术层面的因素，智力题不外乎就是些算法题。其中，举例法、简化推广法、模式匹配法，以及简单构造法可能会特别有用。

面试题目

6.1 有20瓶药丸，其中19瓶装有1克/粒的药丸，余下一瓶装有1.1克/粒的药丸。给你一台称重精准的天平，怎么找出比较重的那瓶药丸？天平只能用一次。（第175页）

6.2 有个8×8棋盘，其中对角的角落上，两个方格被切掉了。给定31块多米诺骨牌，一块骨牌恰好可以覆盖两个方格。用这31块骨牌能否盖住整个棋盘？请证明你的答案（提供范例，或证明为什么不可能）。（第176页）

6.3 有两个水壶，容量分别为5夸脱（美制：1夸脱=0.946升，英制：1夸脱=1.136升）和3夸脱，若水的供应不限量（但没有量杯），怎么用这两个水壶得到刚好4夸脱的水？注意，这两个水壶呈不规则形状，无法精准地装满“半壶”水。（第177页）

6.4 有个岛上住着一群人，有一天来了个游客，定了一条奇怪的规矩：所有蓝眼睛的人都必须尽快离开这个岛。每晚8点会有一个航班离岛。每个人都看得见别人眼睛的颜色，但不知道自己的（别人也不可以告知）。此外，他们不知道岛上到底有多少人是蓝眼睛的，只知道至少有一个人的眼睛是蓝色的。所有蓝眼睛的人要花几天才能离开这个岛？（第177页）

6.5 有栋建筑物高100层。若从第N层或更高的楼层扔下来，鸡蛋就会破掉。若从第N层以下的楼层扔下来则不会破掉。给你2个鸡蛋，请找出N，并要求最差情况下扔鸡蛋的次数为最少。（第178页）

6.6 走廊上有100个关上的储物柜。有个人先是将100个柜子全都打开。接着，每数两个柜子关上一个。然后，在第三轮时，再每隔两个就切换第三个柜子的开关状态（也就是将关上的柜子打开，将打开的关上）。照此规律反复操作100次，在第i轮，这个人会每数i个就切换第i个柜子的状态。当第100轮经过走廊时，只切换第100个柜子的开关状态，此时有几个柜子是开着的？（第179页）

8.7　数学与概率

在面试时碰到的许多数学问题，其中很多看起来像是智力题，其实大都可以运用逻辑、有系统地解决。这些问题通常都以数学或计算机科学为基础，运用这些知识可以解决问题或检验答案对错。本节将介绍那些关系最紧密的数学概念。

1. 素数

大家应该都知道，每一个数都可以分解成素数为乘积。例如：

$$84 = 2^2 * 3^1 * 5^0 * 7^1 * 11^0 * 13^0 * 17^0 * \cdots$$

注意其中不少素数的指数为零。

● 整除

上面的素数定理指出，要想以x整除y（写作$x \backslash y$，或$\mathrm{mod}(y, x) = 0$），x素因子分解的所有素数必须出现在y的素因子分解中。具体如下：

令$x = 2^{j0} * 3^{j1} * 5^{j2} * 7^{j3} * 11^{j4} * \cdots$

8

令$y = 2^{k0} * 3^{k1} * 5^{k2} * 7^{k3} * 11^{k4} * \cdots$

若$x\backslash y$，则$ji <= ki$对所有i都成立。

实际上，x和y的最大公约数为：

$\text{gcd}(x, y) = 2^{\min(j0,\ k0)} * 3^{\min(j1,\ k1)} * 5^{\min(j2,\ k2)} * \cdots$

x和y的最小公倍数为：

$\text{lcm}(x, y) = 2^{\max(j0,\ k0)} * 3^{\max(j1,\ k1)} * 5^{\max(j2,\ k2)} * \cdots$

下面先做一个趣味练习，想一想，将gcd与lcm相乘，结果是什么？

$$\begin{aligned} \text{gcd} * \text{lcm} &= 2^{\min(j0,\ k0)} * 2^{\max(j0,\ k0)} * 3^{\min(j1,\ k1)} >* 3^{\max(j1,\ k1)} * \cdots \\ &= 2^{\min(j0,\ k0) + \max(j0,\ k0)} * 3^{\min(j1,\ k1) + \max(j1,\ k1)} * \cdots \\ &= 2^{j0 + k0} * 3^{j1 + k1} * \cdots \\ &= 2^{j0} * 2^{k0} * 3^{j1} * 3^{k1} * \cdots \\ &= xy \end{aligned}$$

- 素性检查

这个问题很常见，有必要特别说明一下。最原始的做法是从2到$n-1$进行迭代，每次迭代都检查能否整除。

```
boolean primeNaive(int n) {
   if (n < 2) {
     return false;
   }
   for (int i = 2; i < n; i++) {
     if (n % i == 0) {
       return false;
     }
   }
   return true;
}
```

下面有一处很小但重要的改动：只需迭代至n的平方根即可。

```
boolean primeSlightlyBetter(int n) {
   if (n < 2) {
     return false;
   }
   int sqrt = (int) Math.sqrt(n);
   for (int i = 2; i <= sqrt; i++) {
     if (n % i == 0) return false;
   }
   return true;
}
```

使用`sqrt`就够了，因为每个可以整除n的数a，都有个补数b，且$a * b = n$。若$a > sqrt$，则$b < sqrt$（因为$sqrt * sqrt = n$）。因此，当a大于$sqrt$时，就不需要检查了，因为已经用b检查过了。

当然，在现实中，我们真正要做的只是检查n可否被素数整除。这时埃拉托斯特尼筛法（Sieve of Eratosthenes）就派上用场了。

● 生成素数序列：埃拉托斯特尼筛法

埃拉托斯特尼筛法能够非常高效地生成素数序列，原理是剔除所有可被素数整除的非素数。

一开始列出到max为止的所有数字。首先，划掉所有可被2整除的数（2保留）。然后，找到下一个素数（也即下一个不会被划掉的数），并划掉所有可被它整除的数。划掉所有可被2、3、5、7、11等素数整除的数，最终可得到2到max之间的素数序列。

下面是埃拉托斯特尼筛法的实现代码。

```
boolean[] sieveOfEratosthenes(int max) {
   boolean[] flags = new boolean[max + 1];
   int count = 0;

   init(flags); // 将flags中0、1元素除外的所有元素设为true
   int prime = 2;

   while (prime <= max) {
     /* 划掉余下为prime倍数的数字 */
     crossOff(flags, prime);

     /* 找出下一个为true的值 */
     prime = getNextPrime(flags, prime);

     if (prime >= flags.length) {
        break;
     }
   }

   return flags;
}

void crossOff(boolean[] flags, int prime) {
   /* 划掉余下为prime倍数的数字，我们可以从
    * (prime*prime)开始，因为如果k * prime且
    * k < prime，这个值早就在之前的迭代里
    * 被划掉了。 */
   for (int i = prime * prime; i < flags.length; i += prime) {
     flags[i] = false;
   }
}

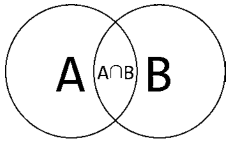
int getNextPrime(boolean[] flags, int prime) {
   int next = prime + 1;
   while (next < flags.length && !flags[next]) {
     next++;
   }
   return next;
}
```

当然，在上面的代码中，还有一些地方可以优化，比如，可以只将奇数放进数组，所需空间即可减半。

2. 概率

概率可以很复杂，还好它是建立在若干基本定理之上，而这些定理可以逻辑推导得出。

下面用韦恩图（Venn diagram）来表示两个事件A和事件B。两个圆圈的区域分别代表事件发生的概率，重叠区域代表事件A与事件B都发生的概率（{A与B都发生}）。



- A与B都发生的概率

假设你朝上面的韦恩图扔飞镖，命中A和B重叠区域的概率有多大？如果你知道命中A的概率，还知道A区域那一块也在B区域中的百分比（也即，命中A的同时也在B区域中的概率），即可用下面的算式计算命中概率：

```
P(A与B都发生) = P(B发生，在A发生的情况下) * P(A发生)
```

举个例子，假设要在1到10（含）之间挑选一个数。挑中一个偶数且这个数在1到5之间的概率有多大？挑中的数在1到5之间的概率为50%，而在1到5之间的数为偶数的概率为40%。因此，两者同时发生的概率为：

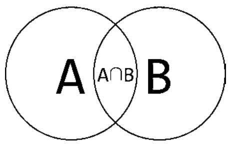
```
P(x为偶数且x <= 5)
    = P(x为偶数，在x <= 5的情况下) * P(x <= 5)
    = (2/5) * (1/2)
    = 1/5
```

- A或B发生的概率

现在，我们又想知道飞镖命中A或B的概率有多大。如果知道单独命中A或B的概率，以及命中两者重叠区域的概率，那么，可以用下面的算式表示命中概率：

```
P(A或B发生) = P(A发生) + P(B发生) - P(A与B都发生)
```

这也很合乎逻辑。只是简单地把两个区域加起来，重叠区域就会被计入两次。我们要减掉一次重叠区域，再次用韦恩图表示：



举个例子，假定我们要在1到10（含）之间挑选一个数。挑中的数为偶数或这个数在1到5之间的概率有多大？显然，挑中一个偶数的概率为50%，挑中的数在1到5之间的概率为50%。两者同时发生的概率为20%，因此前面提到的概率为：

```
P(x为偶数或x <=5)
    = P(x为偶数) + P(x <= 5) - P(x为偶数且x <= 5)
    = (1/2) + (1/2) - (1/5)
    = 4/5
```

掌握上述原理后，理解独立事件和互斥事件的特殊规则就要容易多了。

● 独立

若A与B相互独立（也即，一个事件的发生，推不出另一个事件的发生），那么，P(A与B都发生) = P(A) P(B)。这条规则直接推导自P(B发生，在A发生的情况下) = P(B)，因为A跟B没关系。

● 互斥

若A与B互斥（也即，若一个事件发生，则另一个事件就不可能发生），则P(A或B发生) = P(A) + P(B)。这是因为P(A与B都发生) = 0，所以，删除了之前P(A或B发生)算式中的P(A与B都发生)一项。

奇怪的是，许多人都会搞混独立和互斥的概念。其实两者完全不同。实际上，两个事件不可能同时是独立的又是互斥的（只要两者概率都大于零）。为什么？因为互斥意味着一个事件发生了，另一个事件就不可能发生。而独立是指一个事件的发生跟另一个事件的发生毫无关联。因此，只要两个事件发生的概率不为零，它们就不可能既互斥又独立。

若一个或两个事件的概率为零（也就是不可能发生），那么这两个事件同时既独立又互斥。这很容易直接应用独立和互斥的定义（等式）证明出来。

注意事项

(1) 小心，float和double的精度有别。

(2) 不要假设某个值（比如一条线的斜率）为int型，除非已明确告知这个值为int型。

(3) 除非另有说明，否则不要假定多个事件是独立的（或互斥的）。因此，切忌盲目将概率相乘或相加。

面试题目

7.1 有个篮球框，下面两种玩法可任选一种。

玩法1：一次出手机会，投篮命中得分。

玩法2：三次出手机会，必须投中两次。

如果p是某次投篮命中的概率，则p的值为多少时，才会选择玩法1或玩法2？（第179页）

7.2 三角形的三个顶点上各有一只蚂蚁。如果蚂蚁开始沿着三角形的边爬行，两只或三只蚂蚁撞在一起的概率有多大？假定每只蚂蚁会随机选一个方向，每个方向被选到的几率相等，而且三只蚂蚁的爬行速度相同。

类似问题：在*n*个顶点的多边形上有*n*只蚂蚁，求出这些蚂蚁发生碰撞的概率。（第180页）

7.3 给定直角坐标系上的两条线，确定这两条线会不会相交。（第181页）

7.4 编写方法，实现整数的乘法、减法和除法运算。只允许使用加号。（第182页）

7.5 在二维平面上，有两个正方形，请找出一条直线，能够将这两个正方形对半分。假定正方形的上下两条边与*x*轴平行。（第184页）

7.6 在二维平面上，有一些点，请找出经过点数最多的那条线。（第186页）

7.7 有些数的素因子只有3、5、7，请设计一个算法，找出其中第*k*个数。（第188页）

参考问题：中等难题（#17.11）；高难度题（#18.2）。

8.8　面向对象设计

面向对象设计问题要求求职者设计出类和方法，以实现技术问题或描述真实生活中的对象。这类问题可以让面试官洞悉你的编码风格——至少被认为如此。

这些问题并不那么着重于设计模式，而是意在考察你是否懂得如何打造优雅、容易维护的面向对象代码。若在这类问题上表现不佳，面试可能会亮起红灯。

1. 如何解答面向对象设计问题

对于面向对象设计问题，要设计的对象可能是真实世界的东西，也可能是某个技术任务，不论如何，我们都能以类似的途径解决。以下解题思路适用于很多问题。

- 步骤1：处理不明确的地方

面向对象设计（OOD）问题往往会故意放些烟幕弹，意在检验你是武断臆测，还是提出问题以厘清问题。毕竟，开发人员要是没弄清楚自己要开发什么，就直接挽起袖子开始编码，只会浪费公司的财力物力，还可能造成更严重的后果。

碰到面向对象设计问题时，你应该先问清楚，谁是使用者、他们将如何使用。对某些问题，你甚至还要问清楚“5W1H”，也就是Who（谁）、What（什么）、Where（哪里）、When（何时）、Why（为什么）、How（如何）。

举个例子，假设面试官让你描述咖啡机的面向对象设计。这个问题看似简单明了，其实不然。

这台咖啡机可能是一款工业型机器，设计用来放在大餐厅里，每小时要服务几百位顾客，还要能制作10种不同口味的咖啡。又或者，它可能是设计给老年人使用的简易咖啡机，只要能制作简单的黑咖啡就行。这些用例将大大影响你的设计。

- 步骤2：定义核心对象

了解我们要设计的东西后，接下来就该思考系统的“核心对象”了。比如，假设要为一家餐馆进行面向对象设计。那么，核心对象可能包括餐桌（`Table`）、顾客（`Guest`）、宴席（`Party`）、订单（`Order`）、餐点（`Meal`）、员工（`Employee`）、服务员（`Server`）和领班（`Host`）。

- 步骤3：分析对象关系

定义出核心对象之后，接下来要分析这些对象之间的关系。其中，哪些对象是其他对象的数据成员？哪个对象继承自别的对象？对象之间是多对多的关系，还是一对多的关系？

比如，在处理餐馆问题时，我们可能会想到以下设计。

- 宴席包括很多顾客。
- 服务员和领班都继承自员工。
- 每一张餐桌对应一个宴席，但每个宴席可能拥有多张餐桌。
- 每家餐馆有一个领班。

分析对象关系一定要非常小心——我们经常会作出错误假设。比如，哪怕是一张餐桌也可能包含多个“宴席”（在热门餐馆里，“拼桌”很常见）。进行设计时，你应该跟面试官探讨一下，了解你的设计要做到多通用。

- 步骤4：研究对象的动作

到这一步，你的面向对象设计应该初具雏形了。接下来，该想想对象可执行的关键动作，以及对象之间的关联。你可能会发现自己遗漏了某些对象，这时就需要补全并更新设计。

例如，一个“宴席”对象（由一群顾客组成）走进了“餐馆”，一位“顾客”找“领班”要求一张“餐桌”。“领班”开始查验“预订”（Reservation），若找到记录，便将“宴席”对象领到“餐桌”前。否则，“宴席”对象就要排在列表末尾。等到其他“宴席”对象离开后，有“餐桌”空出来，就可以分配给列表中的“宴席”对象。

2. 设计模式

因为面试官想要考察的是你的能力而不是知识，大部分面试都不会考设计模式。不过，掌握单例设计模式（Singleton）和工厂方法（Factory Method）设计模式对面试来说特别有用，所以，接下来我们会作简单介绍。

设计模式太多了，限于篇幅，没办法在本书中一一探讨。你可以挑本专门讨论这个主题的书来研读，这对提高你的软件工程技能会大有裨益。

- 单例设计模式

单例设计模式确保一个类只有一个实例，并且只能通过类内部方法访问此实例。当你有个“全局”对象，并且只会有一个这种实例时，这个模式特别好用。比如，在实现“餐馆”时，我们可能想让它只有一个“餐馆”实例。

```
public class Restaurant {
   private static Restaurant _instance = null;
   protected Restaurant() { ... }
   public static Restaurant getInstance() {
     if (_instance == null) {
        _instance = new Restaurant();
     }
     return _instance;
   }
 }
```

- 工厂方法设计模式

工厂方法提供接口以创建某个类的实例，由子类决定实例化哪个类。实现时，你可以将创建器类（Creator）设计为抽象类型，不为工厂方法提供具体实现；或者，创建器类是实体类，为工厂方法提供具体实现。在这种情况下，工厂方法需要传入参数，代表该实例化哪个类。

```
public class CardGame {
   public static CardGame createCardGame(GameType type) {
     if (type == GameType.Poker) {
        return new PokerGame();
     } else if (type == GameType.BlackJack) {
        return new BlackJackGame();
     }
     return null;
   }
}
```

面试题目

8.1 请设计用于通用扑克牌的数据结构。并说明你会如何创建该数据结构的子类，实现“二十一点”游戏。（第192页）

8.2 设想你有个呼叫中心，员工分成三个层级：接线员、主管和经理。客户来电会先分配给有空的接线员。若接线员处理不了，就必须将来电往上转给主管。若主管没空或是无法处理，则将来电往上转给经理。请设计这个问题的类和数据结构，并实现一个`dispatchCall()`方法，将客户来电分配给第一个有空的员工。（第195页）

8.3 运用面向对象原则，设计一款音乐点唱机。（第198页）

8.4 运用面向对象原则，设计一个停车场。（第200页）

8.5 请设计在线图书阅读器系统的数据结构。（第203页）

8.6 实现一个拼图程序。设计相关数据结构并提供一种拼图算法。假设你有一个`fitsWith`方法，传入两块拼图，若两块拼图能拼在一起，则返回`true`。（第207页）

8.7 请描述该如何设计一个聊天服务器。要求给出各种后台组件、类和方法的细节，并说明其中最难解决的问题会是什么。（第210页）

8.8 “奥赛罗棋”（黑白棋）的玩法如下：每一枚棋子的一面为白，一面为黑。游戏双方各执黑、白棋子对决，当一枚棋子的左右或上下同时被对方棋子夹住，这枚棋子就算是被吃掉了，随即翻面为对方棋子的颜色。轮到你落子时，必须至少吃掉对方一枚棋子。任意一方无子可落时，游戏即告结束。最后，棋盘上棋子较多的一方获胜。请运用面向对象设计方法，实现“奥赛罗棋”。（第214页）

8.9 设计一种内存文件系统（in-memory file system）的数据结构和算法，并说明具体做法。如有可行，请用代码举例说明。（第217页）

8.10 设计并实现一个散列表，使用链接（即链表）处理碰撞冲突。（第219页）

参考问题：线程与锁（#16.3）。

8.9　递归和动态规划

尽管递归问题五花八门，但题型大都类似。一个问题是不是递归的，就看它能不能分解为子问题进行求解。

当你听到问题是这么开头的：“设计一个算法，计算第n个……”，“编写代码列出前n个……”，“实现一个方法，计算所有……”等等，那么，这基本上就是一个递归问题。

熟能生巧！练习的越多，就越容易识别递归问题。

1. 解决之道

递归的解法，根据定义，就是从较小的子问题逐渐逼近原始问题。很多时候，只要在$f(n-1)$的解法中加入、移除某些东西或者稍作修改就能算出$f(n)$。而在其他情况下，答案可能更为复杂。

你应该双管齐下，自下而上和自上而下两种递归解法都要考虑。简单构造法对递归问题就很奏效。

● 自下而上的递归

自下而上的递归往往最为直观。首先要知道如何解决简单情况下的问题，比如，只有一个元素的列表，找出有两个、三个元素的列表的解法，依此类推。这种解法的关键在于，如何从先前解出来的答案，构建出后续情况的答案。

● 自上而下的递归

自上而下的递归可能比较复杂，不过对某些问题很有必要。遇到这类问题时，我们要思考如何才能将情况N下的问题分解成多个子问题。同时要注意子问题是否重叠了。

2. 动态规划

在面试中，动态规划（Dynamic programming，DP）问题很少问及，原因很简单，短短45分钟的面试要解决这类问题实在太难了。就算是出色的求职者，面对这类问题通常也难有上佳表现，因此动态规划问题不适合用来评估求职者。

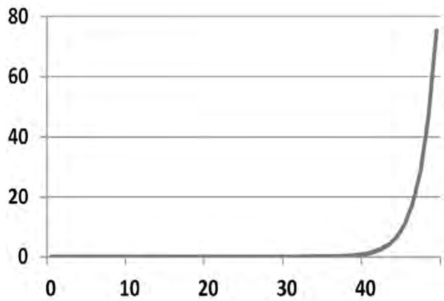
要是不走运，碰到了动态规划问题，你可以用跟递归问题差不多的解决方法来处理。区别在于，中间结果要“缓存”起来，以备后续使用。

● 动态规划法简单示例：斐波那契数列

下面举个动态规划法的简单例子。想象一下，面试官要求你实现一个程序，生成斐波那契数列的第n项数字。听起来很简单，对吧？

```
int fibonacci(int i) {
   if (i == 0) return 0;
   if (i == 1) return 1;
   return fibonacci(i - 1) + fibonacci(i - 2);
}
```

这个函数要用多长时间执行？计算斐波那契数列第n项要先算出前面的n–1项。而每次函数调用又包含两次递归调用，也就是说，执行时间为$O(2^n)$。下面的图表显示在普通台式机上的执行结果，执行时间呈指数级上升。



生成第N项斐波那契数所需秒数

只要对上面的函数稍作修改，就可以将时间复杂度优化为$O(N)$。具体做法就是将每次调用`fibonacci(i)`的结果“缓存”起来。

```
int[] fib = new int[max];
int fibonacci(int i) {
   if (i == 0) return 0;
   if (i == 1) return 1;
   if (fib[i] != 0) return fib[i]; // 返回先前缓存的结果
   fib[i] = fibonacci(i - 1) + fibonacci(i - 2); // 缓存结果
   return fib[i];
}
```

在普通电脑上，之前的递归版本生成第50项斐波那契数用时可能超过一分钟，而动态规划方法只需几毫秒就能产生第10 000项斐波那契数。当然，若采用上面这段代码，`int`型变量很快就会溢出。

如你所见，动态规划法没什么好怕的。只不过要缓存中间结果，比递归稍稍复杂些。解决这类问题，有个好办法就是先以一般的递归法实现，然后添加缓存部分。

3. 递归和迭代解法

递归算法的空间效率很低。每次递归调用都会在栈上增加一层，也就是说，若算法包含$O(n)$次递归调用，就要使用$O(n)$内存。不得了！

所有的递归代码都能改为迭代式的实现，尽管有时候这么做代码会复杂得多。在一头扎入递归代码之前，先问问自己用迭代方式实现会有多难，并与面试官讨论不同解法的优劣差异。

面试题目

9.1 有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上1阶、2阶或3阶。实现一个方法，计算小孩有多少种上楼梯的方式。（第221页）

9.2 设想有个机器人坐在$X×Y$网格的左上角，只能向右、向下移动。机器人从(0,0)到(X,Y)有多少种走法？

进阶

假设有些点为“禁区”，机器人不能踏足。设计一种算法，找出一条路径，让机器人从左上角移动到右下角（第222页）。

9.3 在数组`A[0...n-1]`中，有所谓的魔术索引，满足条件`A[i] = i`。给定一个有序整数数组，元素值各不相同，编写一个方法，在数组`A`中找出一个魔术索引，若存在的话。

进阶

如果数组元素有重复值，又该如何处理？（第224页）

9.4 编写一个方法，返回某集合的所有子集。（第226页）

9.5 编写一个方法，确定某字符串的所有排列组合。（第229页）

9.6 实现一种算法，打印n对括号的全部有效组合（即左右括号正确配对）。

示例

输入：3

输出：((())), (()()), (())(), ()(()), ()()()（第230页）

9.7 编写函数，实现许多图片编辑软件都支持的“填充颜色”功能。给定一个屏幕（以二维数组表示，元素为颜色值）、一个点和一个新的颜色值，将新颜色值填入这个点的周围区域，直到原来的颜色值全都改变。（第232页）

9.8 给定数量不限的硬币，币值为25分、10分、5分和1分，编写代码计算n分有几种表示法。（第232页）

9.9 设计一种算法，打印八皇后在8×8棋盘上的各种摆法，其中每个皇后都不同行、不同列，也不在对角线上。这里的“对角线”指的是所有的对角线，不只是平分整个棋盘的那两条对角线。（第234页）

9.10 给你一堆n个箱子，箱子宽w_i、高h_i、深d_i。箱子不能翻转，将箱子堆起来时，下面箱子的宽度、高度和深度必须大于上面的箱子。实现一个方法，搭出最高的一堆箱子，箱堆的高度为每个箱子高度的总和。（第236页）

9.11 给定一个布尔表达式，由`0`、`1`、`&`、`|`和^等符号组成，以及一个想要的布尔结果`result`，实现一个函数，算出有几种括号的放法可使该表达式得出`result`值。

示例

表达式：`1^0|0|1`

期望结果：`false(0)`

输出：`1^((0|0)|1)`和`1^(0|(0|1))`两种方式（第238页）

参考问题：链表（#2.2、#2.5、#2.7）；栈与队列（#3.3）；树与图（#4.1、#4.3、#4.4、#4.5、#4.7、#4.8、#4.9）；位操作（#5.7）；智力题（#6.4）；排序与查找（#11.5、#11.6、#11.7、#11.8）；C和C++（#13.7）；中等难题（#17.13、#17.14）；高难度题（#18.4、#18.7、#18.12、#18.13）。

8.10　扩展性与存储限制

扩展性面试题看似吓人，其实这类问题算得上是最简单的。它们不会暗藏什么“陷阱”，不会有什么花招，也不需要花哨的算法——至少通常不会有。你不需要学习分布式系统方面的课程，也不必具备系统设计的相关经验。只要稍加练习，任何心思缜密且够聪明的软件工程师都能轻松搞定这些问题。

1. 循序渐进法

面试官并不是想考察你掌握了多少系统设计知识；实际上，除了考察最基本的计算机科学概念，面试官一般不会考具体的知识点。相反，他们想要评估的是你分解棘手问题的能力，以及用所学知识解决问题的能力。以下这些步骤有助于应对大多数系统设计问题。

- 步骤1：大胆假设

假设一台计算机就能装下全部数据，且存储上没有任何限制。你会如何解决问题？由此得出的答案，可以为你最终解决问题提供基本思路。

- 步骤2：切合实际

现在，让我们回到问题本身。一台计算机究竟能装下多少数据，拆分这些数据会产生什么问

题？通常，我们需要考虑如何合理拆分数据，以及一台计算机需要不同的数据片段时，如何得知该去哪里查找，等等。

- 步骤3：解决问题

最后，想一想该如何处理步骤2发现的问题。请记住，这些解决方案应该能彻底消除这些问题，或至少改善一下状况。通常情况下，你可以继续使用（进行一定修改）步骤1描述的方法，但偶尔也需要改弦易张，从根本上改变解决方案。

请注意，迭代法通常很有用。也就是说，等你解决好步骤2发现的问题，可能又会冒出新问题，你还要着手处理这些新问题。

你的目标不是重新设计公司耗资数百万美元搭建的复杂系统，而是证明你有能力分析和解决问题。检验自己的解法，四处挑错并予以修正，是个向面试官展现实力的不错方法。

2. 你需要知道的：信息、策略与问题

- 典型系统

尽管仍有公司在使用大型机，可大多数互联网公司还是喜欢使用由普通计算机互联组成的大型系统。通常情况下，你可以假定自己就是在使用这种系统。

面试之前，你最好填写下面的表格。利用这张表，可以估算出一台计算机可存储多少数据。

组　　件	一般容量/数值
硬盘空间	
内存	
网络传输延迟	

- 拆分大量的数据

尽管有时我们可以增加计算机的硬盘空间，不过，难免会遇到必须将数据拆分至多台计算机的情形。随之而来的问题是，哪些数据要放在哪一台机器上。下面有几种策略可供参考。

- ❑ 按出现的顺序

我们可以按出现的顺序直接划分数据。也就是说，有新数据进来时，先放进当前机器，填满之后，再加一台机器。这么做的好处是不会浪费资源。然而，根据具体问题和数据集的不同，查找表可能会变得非常复杂、异常巨大。

- ❑ 按散列值

另一种做法是根据数据的散列值存放数据。具体一点来说，我们会采取以下步骤：(1) 根据数据挑选某种键；(2) 利用散列函数得到键的散列值；(3) 将散列值除以机器数量求得余数；(4) 将数据存储在这个值对应的机器上。也就是说，数据会存放在编号为`#[mod(hash(key), N)]`的机器上。

这种做法的好处是不用创建数据查找表。每一台计算机自动掌握数据的存储位置。然而，这也会出问题，那就是某台机器的数据可能会多一些，并最终超出它的存储容量。若发生这种情况，可以将数据迁移到其他机器上，以实现更好的负载均衡（但开销很大），或者将这台机器的数据拆分到两台机器上（形成一组树状结构的机器）。

❑ 按实际值

按散列值划分数据本质上是随机的；数据代表的具体意义与存储数据的机器之间，并不存在任何关系。在某些情况下，我们也许可以利用数据所代表的信息来降低系统延迟。

例如，假设你正在设计一个社交网站。虽然人们的朋友会来自世界各地，但实际上，相比俄罗斯普通公民，住在墨西哥的人可能拥有更多来自墨西哥的朋友。或许，我们可以将"类似"数据存储在同一台机器上，这样在查找墨西哥人的朋友时，只需访问较少数量的机器就能取得相关资料。

❑ 随机存储

通常情况下，我们只是随机划分数据，再实现一个查找表以表明哪台机器拥有哪些数据。虽然这肯定需要一张巨大的查找表，但它简化了系统设计的某些方面，使我们得以实现更好的负载均衡。

3. 示例：查找所有包含某一组词的文件

给定数百万份文件，如何找出所有包含某一组词的文件？我们不关心这些词出现的顺序，但它们必须是完整的单词。也就是说，"book"与"bookkeeper"不是一回事。

在着手解决问题之前，我们需要考虑`findWords`程序只用一次，还是要反复调用。假设需要多次调用`findWords`程序来扫描这些文件，那么，我们可以接受预处理的开销。

● 步骤1

第一步是先忘记我们有数以百万计的文件，假装只有几十个文件。在这种情况下，如何实现`findWords`呢？（提示：不要急着看下文，先试着自己解解看。）

一种方法是预处理每个文件，并创建一个散列表的索引。这个散列表会将词映射到含有这个词的一组文件。

```
"books" -> {doc2, doc3, doc6, doc8}
"many" -> {doc1, doc3, doc7, doc8, doc9}
```

若要查找"many books"，只需对"books"和"many"的值进行交集运算，于是得到结果`{doc3, doc8}`。

● 步骤2

现在，回到最初的问题。若有数百万份文件，会有什么问题？首先，我们可能需要将文件分散到多台机器上。此外，我们还要考虑很多因素，比如要查找的单词数量、在文件中重复出现的次数等，一台机器可能放不下完整的散列表。假设我们就要按这个限制进行设计。

文件分散到多台机器上会引出以下几个很关键的关注点。

(1) 如何划分该散列表？我们可以按关键字划分，例如，某台机器上存放有包含某个单词的全部文件。或者，可以按文件来划分，这样一台机器上只会存放对应某个关键字的部分文件，而非全部。

(2) 一旦决定了如何划分数据，我们可能需要在一台机器上对文件进行处理，并将结果推送到其他机器上。这个过程会是什么呢？（注意：若按文件划分散列表，这一步可能就没有必要。）

(3) 我们需要找到一种方法获知哪台机器拥有哪些数据。这个查找表会是什么样的？又该存储在什么地方？

这只是其中三个，可能还会有更多其他的关注点。

● 步骤3

在步骤3中，我们要找出解决这些关注点的解决方案。其中一种解法是按字母顺序划分不同的关键字，这样，每台机器便可以处理一串词。例如，从“after”直到“apple”。

我们可以实现一个简单的算法，按字母顺序遍历所有关键字，并尽可能多地将数据存储在一台机器上。当这台机器的空间被占满之后，我们便转到下一台机器。

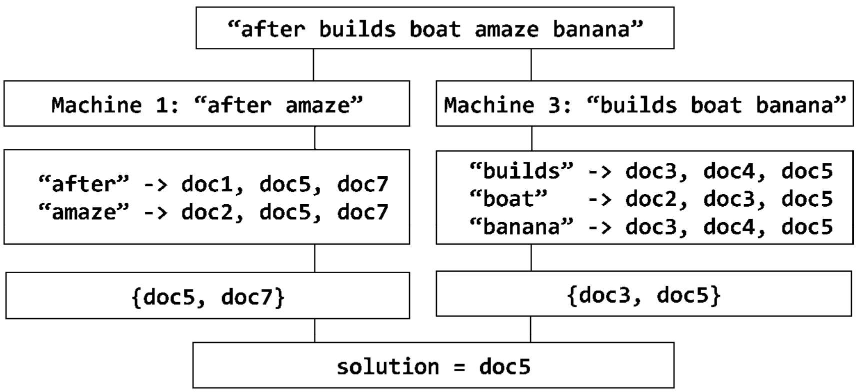
这种方法的优点是查找表会比较小而且简单（因为它只需包含一系列指定的值），每台机器可存储一份查找表的副本。然而，不足之处在于新增文件或单词时，我们可能需要改变关键字的位置，这么做开销很大。

为了找到匹配某一组字符串的所有文件，我们会先对这一组字符串进行排序，然后给每一台机器发送与字符对应的查找请求。例如，若待查字符串为“after builds boat amaze banana”，一号机器就会接收到查找{“after”，“amaze”}的请求。

一号机器开始查找包含“after”与“amaze”的文件，并对这些文件执行交集运算。三号机器则处理{“banana”，“boat”，“builds”}这几个关键字，同样也会对文件进行交集运算。

最后，发送请求的机器再对一号机器及三号机器返回的结果作交集运算。

下图描述了整个过程。



面试题目

10.1 假设你正在搭建某种服务，有多达1000个客户端软件会调用该服务，取得每天盘后股票价格信息（开盘价、收盘价、最高价与最低价）。假设你手里已有这些数据，存储格式可自行定义。你会如何设计这套面向客户端的服务，向客户端软件提供信息？你将负责该服务的研发、部署、持续监控和维护。描述你想到的各种实现方案，以及为何推荐采用你的方案。该服务的实现技术可任选，此外，可以选用任何机制向客户端分发信息。（第241页）

10.2 你会如何设计诸如Facebook或LinkedIn的超大型社交网站？请设计一种算法，展示两个人之间的“连接关系”或“社交路径”（比如，我 → 鲍勃 → 苏珊 → 杰森 → 你）。（第243页）

10.3 给定一个输入文件，包含40亿个非负整数，请设计一种算法，产生一个不在该文件中的整数。假定你有1GB内存来完成这项任务。

进阶

如果只有10MB内存可用，该怎么办？假定所有值都是唯一的。（第240页）

10.4 给定一个数组，包含1到*N*的整数，*N*最大为32 000，数组可能含有重复的值，且*N*的取值不定。若只有4KB内存可用，该如何打印数组中所有重复的元素。（第248页）

10.5 如果要设计一个网络爬虫程序，该怎么避免陷入无限循环？（第249页）

10.6 给定100亿个网址，如何检测出重复的文件？这里所谓的”重复“是指两个URL完全相同。（第250页）

10.7 想象有个Web服务器，实现简化版搜索引擎。这套系统有100台机器来响应搜索查询，可能会对另外的机器集群调用`processSearch(string query)`以得到真正的结果。响应查询请求的机器是随机挑选的，因此两个同样的请求不一定由同一台机器响应。方法`processSearch`的开销很大，请设计一种缓存机制，缓存最近几次查询的结果。当数据发生变化时，务必说明该如何更新缓存。（第251页）

参考问题：面向对象设计（#8.7）。

8.11 排序与查找

花时间学习掌握常见的排序和查找算法，回报巨大，很多排序与查找问题，实际上只是将大家熟悉的算法稍作修改而已。因此，处理这类问题的诀窍就是逐一考虑各种不同的排序算法，看看哪一种特别合适。

举个例子，假设你被问到如下问题：给定一个含有`Person`对象且非常大的数组，请按年龄从小到大对数组元素进行排序。

根据题目，有以下两点值得注意：

(1) 数组很大，所以效率非常重要；

(2) 根据年龄排序，所以这些数值的范围较小。

检查各种排序算法，可能会注意到“桶排序”（或称基数排序），特别适用于这个问题。事实上，我们用到的桶子数目并不多（一个年龄对应一个），最终执行时间为$O(n)$。

1. 常见的排序算法

学习（或复习）常见的排序算法是提升自身水平的绝佳方式。下面介绍的五种算法中，归并排序（Merge Sort）、快速排序（Quick Sort）和基数排序（Radix Sort）是面试中最常用的三种。

8

- 冒泡排序|执行时间：平均情况与最差情况为$O(n^2)$，存储空间：$O(1)$

冒泡排序（Bubble Sort）是先从数组第一个元素开始，依次比较相邻两个数，若前者比后者大，就将两者交换位置，然后处理下一对，依此类推，不断扫描数组，直到完成排序。

● 选择排序|执行时间：平均情况与最差情况为$O(n^2)$，存储空间：$O(1)$

选择排序（Selection Sort）有点“小儿科”：简单而低效。我们会线性逐一扫描数组元素，从中挑出最小的元素，将它移到最前面（也就是与最前面的元素交换）。然后，再次线性扫描数组，找到第二小的元素，并移到前面。如此反复，直到全部元素各归其位。

● 归并排序|执行时间：平均情况与最差情况为$O(n \log(n))$，存储空间：看情况

归并排序是将数组分成两半，这两半分别排序后，再归并在一起。排序某一半时，继续沿用同样的排序算法，最终，你将归并两个只含一个元素的数组。这个算法的重担都落在“归并”的部分上。

在下面的代码中，merge方法会将目标数组的所有元素拷贝到临时数组helper中，并记下数组左、右两半的起始位置（helperLeft和helperRight）。然后，迭代访问helper数组，将左右两半中较小的元素，复制到目标数组中。最后，再将余下所有元素复制回目标数组。

```
void mergesort(int[] array, int low, int high) {
   if (low < high) {
     int middle = (low + high) / 2;
     mergesort(array, low, middle); // 排序左半部分
     mergesort(array, middle + 1, high); // 排序右半部分
     merge(array, low, middle, high); // 归并
   }
}

void merge(int[] array, int low, int middle, int high) {
   int[] helper = new int[array.length];

   /* 将数组左右两半拷贝到helper数组中 */
   for (int i = low; i <= high; i++) {
     helper[i] = array[i];
   }

   int helperLeft = low;
   int helperRight = middle + 1;
   int current = low;

   /* 迭代访问helper数组。比较左、右两半的元素，
    * 并将较小的元素复制到原先的数组中。
    */
   while (helperLeft <= middle && helperRight <= high) {
     if (helper[helperLeft] <= helper[helperRight]) {
        array[current] = helper[helperLeft];
        helperLeft++;
     } else { // 如果右边的元素小于左边的元素
        array[current] = helper[helperRight];
        helperRight++;
     }
     current++;
   }

   /* 将数组左半部分剩余元素
    * 复制到目标数组中 */
   int remaining = middle - helperLeft;
```

```
  for (int i = 0; i <= remaining; i++) {
    array[current + i] = helper[helperLeft + i];
  }
}
public static void mergesort(int[] array) {
    int[] helper = new int[array.length];
    mergesort(array, helper, 0, array.length - 1);
}
```

你可能会发现，上述代码只是将helper数组左半部分剩余元素，复制回目标数组中。为什么不复制右半部分的呢？那是因为这部分元素早已在目标数组中，无需复制。

下面以数组[1, 4, 5 || 2, 8, 9]（符号“||”表示分界点）为例进行说明。在合并左右两部分的元素之前，helper数组与目标数组末尾都是[8, 9]。将4个元素（1、4、5和2）复制到目标数组时，[8, 9]仍在原处。所以，也就不需要复制这两个元素。

- 快速排序|执行时间：平均情况为$O(n \log(n))$，最差情况为$O(n^2)$，存储空间：$O(\log(n))$

快速排序是随机挑选一个元素，对数组进行分割，以将所有比它小的元素排在前面，比它大的元素则排在后面。这里的分割经由一系列元素交换的动作完成（见下文）。

如果我们根据某元素再对数组（及其子数组）进行分割，并反复执行，最后数组就会变成有序的。然而，因为无法确保分割元素就是数组的中位数（或接近中位数），快速排序的效率可能会非常低下，这也是为什么最差情况时间复杂度为$O(n^2)$的原因。

```
void quickSort(int arr[], int left, int right) {
   int index = partition(arr, left, right);
   if (left < index - 1) { // 排序左半部分
     quickSort(arr, left, index - 1);
   }
   if (index < right) { // 排序右半部分
     quickSort(arr, index, right);
   }
}

int partition(int arr[], int left, int right) {
   int pivot = arr[(left + right) / 2]; // 挑出一个基准点
   while (left <= right) {
     // 找出左边中应被放到右边的元素
     while (arr[left] < pivot) left++;

     // 找出右边中应被放到左边的元素
     while (arr[right] > pivot) right--;

     // 交换元素，同时调整左右索引值
     if (left <= right) {
        swap(arr, left, right); // 交换元素
        left++;
        right--;
     }
   }
   return left;
}

```

● 基数排序|执行时间：$O(kn)$（见下文）

基数排序是个整数（或其他一些数据类型）排序算法，充分利用整数的位数有限这一事实。使用基数排序时，我们会迭代访问数字的每一位，按各个位对这些数字分组。比如说，假设有一个整数数组，我们可以先按个位对这些数字进行分组，于是，个位为0的数字就会分在同一组里。然后，再按十位进行分组，如此反复执行同样的过程，逐级按更高位进行排序，直到最后整个数组变为有序数组。

其他比较算法的平均情况执行时间不会优于$O(n \log(n))$，相比之下，基数排序的执行时间为$O(kn)$，其中n为元素个数，k为数字的位数。

2. 查找算法

提到查找算法时，我们一般都会想到二分查找法。这个算法的确非常有用，值得研习。在二分查找中，要在有序数组里查找元素x，我们会先取数组中间元素与x作比较。若x小于中间元素，则搜索数组的左半部。若x大于中间元素，则搜索数组的右半部。然后，重复这个过程，将左半部和右半部视作子数组继续搜索。我们再次取这个子数组的中间元素与x作比较，然后搜索左半部或右半部。我们会重复这一过程，直至找到x或子数组大小为0。

概念上似乎很简单，但要真正掌握全部细节，却比你想象的还要困难。研读以下代码时，请注意哪里要加1、哪里要减1。

```
int binarySearch(int[] a, int x) {
   int low = 0;
   int high = a.length - 1;
   int mid;

   while (low <= high) {
     mid = (low + high) / 2;
     if (a[mid] < x) {
        low = mid + 1;
     } else if (a[mid] > x) {
        high = mid - 1;
     } else {
        return mid;
     }
   }
   return -1; // 错误
}

int binarySearchRecursive(int[] a, int x, int low, int high) {
   if (low > high) return -1; // 错误

   int mid = (low + high) / 2;
   if (a[mid] < x) {
     return binarySearchRecursive(a, x, mid + 1, high);
   } else if (a[mid] > x) {
     return binarySearchRecursive(a, x, low, mid - 1);
   } else {
     return mid;
   }
}
```

除了二分查找法，还有很多种查找数据结构的方法，总之，我们不要拘泥于二分查找法。比如说，你可以利用二叉树或使用散列表来查找某结点。尽情开拓思路吧！

面试题目

11.1 给定两个排序后的数组A和B，其中A的末端有足够的缓冲空容纳B。编写一个方法，将B合并入A并排序。（第255页）

11.2 编写一个方法，对字符串数组进行排序，将所有变位词排在相邻的位置。（第256页）

11.3 给定一个排序后的数组，包含n个整数，但这个数组已被旋转过很多次，次数不详。请编写代码找出数组中的某个元素。可以假定数组元素原先是按从小到大的顺序排列的。
示例
输入：在数组{15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}中找出5。
输出：8（元素5在该数组中的索引）（第257页）

11.4 设想你有个20GB的文件，每一行一个字符串。请说明将如何对这个文件进行排序。（第258页）

11.5 有个排序后的字符串数组，其中散布着一些空字符串，编写一个方法，找出给定字符串的位置。
示例
输入：在字符串数组{“at”, “”, “”, “”, “ball”, “”, “”, “car”, “”, “”, “dad”, “”, “”}中，查找“ball”。
输出：4 （第259页）

11.6 给定$M \times N$矩阵，每一行、每一列都按升序排列，请编写代码找出某元素。（第260页）

11.7 有个马戏团正在设计叠罗汉的表演节目，一个人要站在另一人的肩膀上。出于实际和美观的考虑，在上面的人要比下面的人矮一点、轻一点。已知马戏团每个人的高度和重量，请编写代码计算叠罗汉最多能叠几个人。
示例
输入(ht, wt)：(65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)
输出：从上往下数，叠罗汉最多能叠6层：(56, 90) (60,95) (65,100) (68,110) (70,150) (75,190) （第265页）

11.8 假设你正在读取一串整数。每隔一段时间，你希望能找出数字x的秩（小于或等于x的值的数目）。请实现数据结构和算法支持这些操作，也就是说，实现track(int x)方法，每读入一个数字都会调用该方法；以及getRankOfNumber(int x)方法，返回值为小于或等于x的元素个数（不包括x本身）。
示例
数据流为（按出现的先后顺序）：5, 1, 4, 4, 5, 9, 7, 13, 3
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1

8

`getRankOfNumber(4) = 3` （第267页）

参考问题：数组与字符串（#1.3）；递归（#9.3）；中等难题（#17.6、#17.12）；高难度题（#18.5）。

8.12　测试

在念叨着“我又不是测试员”准备跳过本章之前，请你三思。对于软件工程师来说，测试是项很重要的工作，因此，在面试中你很可能会碰到测试问题。当然，如果你刚好要应聘测试职位（或软件测试工程师），那就更应该好好研读本章。

测试问题一般分为以下四类：(1) 测试现实生活中的事物（比如一支笔）；(2) 测试一套软件；(3) 编写代码测试一个函数；(4) 调试解决已知问题。针对每一类题型，我们都会给出相应的解法。

请记住，处理这四类问题时，切勿假设使用者会好好地正常操作。请做好应对用户误用乱用软件的准备。

1. 面试官想考察什么

表面上看，测试问题主要考察你能否想到周全完备的测试用例。这在某种程度上也是对的，求职者确实需要想出一系列合理的测试用例。

但除此之外，面试官还想考察以下几个方面。

- **全局观**：你是否真的了解软件是怎么回事？你能否正确区分测试用例的优先顺序？比如说，假设你被问到该如何测试像亚马逊这样的电子商务系统。若能确保产品图片显示位置正确，当然也不错，但最重要的还是确保支付流程万无一失，货品能顺利地进入发货流程，并且顾客绝对不能被重复扣款。
- **懂整合**：你是否了解软件的工作原理？该如何将它们整合成更大的软件生态系统？假设要测试谷歌电子表格（Spreadsheets），你自然会想到测试文档的打开、存储及编辑功能。但是，实际上，谷歌电子表格也是大型软件生态系统的重要组成部分之一。所以，你还需将它与Gmail、各种插件和其他模块整合在一起进行测试。
- **会组织**：你能否有条有理地处理问题？还是处理问题时毫无条理？被要求提出照相机的测试用例时，有些求职者只会一股脑儿倒出一些杂乱无章的想法。而优秀的求职者却能将测试功能分为几类，比如拍照、照片管理、设置，等等。在创建测试用例时，这种结构化处理方法还有助于你将工作做得更周全。
- **可操作**：你制定的测试计划是否合理，行之有效？比如，如有用户报告，软件会在打开某张图片时崩溃，而你却只是要求他们重新安装软件，这显然太不实际了。你的测试计划必须切实可行，便于公司操作落实 。

倘若能在面试中充分展现这些方面，那么，你无疑就是任何测试团队所梦寐以求的重要一员。

2. 测试现实生活中的事物

问及该如何测试一支笔时，有些求职者会感到莫名惊诧。毕竟，你要测试的不是软件吗？没错，但这些关于“现实生活”的问题其实很常见。我们先来看看下面这个例子吧！

比如有这么一个问题：如何测试一枚回形针？

- 步骤1：使用者是哪些人？做什么用？

你需要跟面试官讨论一下，谁会使用这个产品，做什么用。回答可能出乎你的意料，比如，回答可能是“老师，把纸张夹在一起”或“艺术家，为了弯成动物的造型”。又或者，两者皆要考虑。这个问题的答案，将影响你怎么处理后续问题。

- 步骤2：有哪些用例？

列出回形针的一系列用例，这对解决问题很有帮助。在这个例子中，用例可能是，将纸张固定在一起，且不得破坏纸张。

若是其他问题，可能会有多个用例。比如，某产品要能够发送和接收内容，或擦写和删除功能，等等。

- 步骤3：有哪些使用限制？

使用限制可能是，回形针一次可以夹最多30张纸，且不会造成永久性损害（比如弯掉），另外，可以夹30到50张纸时，只不过会发生轻微变形。

同时，使用限制也要考虑环境因素。比如，回形针可否在非常温暖的环境下（33 ~ 43摄氏度）使用？在极寒环境下呢？

- 步骤4：压力与失效情况下的状态如何？

没有产品是万无一失的，所以，在测试中，还必须分析失效情况。跟面试官探讨时，最好问一下在什么情况下产品失效是可接受的（甚至是必要的），以及什么样才算是失效。

举个例子，要你测试一台洗衣机，你可能会认为洗衣机至少要能洗30件T恤衫或裤子。一次放进30到45件衣服，可能会导致轻微失效，因为衣物洗得不够干净。若超过45件衣物，出现极端失效或许可以接受。不过，这里所谓的极端失效，应该是指洗衣机根本不该进水，绝对不应该让水溢出来或引发火灾。

- 步骤5：如何执行测试？

有些情况下，讨论执行测试的细节可能很重要。比如，若要确保一把椅子能正常使用5年，你恐怕不会把它放在家里，等上5年再来看结果。相反，你需要定义何谓“正常”使用情况（每年会在椅子上坐多少次？扶手呢？）。然后，除了做一些手动测试，你可能还会想到找台机器，自动执行某些功能测试。

3. 测试一套软件

测试软件与测试现实生活的事物非常相似。主要差异在于，软件测试往往更强调执行测试的细节。

请注意，软件测试主要有如下两个方面。

- **手动测试与自动化测试**：理想情况下，我们当然希望能够自动化所有的测试工作，不过这不太现实。有些东西还是手动测试来的更好，因为某些功能对计算机而言过于定性，计算机很难有效地检查（比如，内容带有淫秽色情成分）。此外，计算机只能机械地识别明确告知过的情况，而人类就不一样了，通过观察可能发现亟待验证的新问题。因此，在测试过程中，无论是人工还是计算机，两者都不可或缺。

- **黑盒测试与白盒测试**：两者的区别反映了我们对软件内部机制的掌控程度。在黑盒测试中，我们只关心软件的表象，并且仅测试其功能。而在白盒测试中，我们会了解程序的内部机制，还可以分别对每一个函数单独进行测试。我们也可以自动执行部分黑盒测试，只不过难度要大得多。

下面介绍一种测试方法，并从头到尾细述一遍。

- 步骤1：要做黑盒测试还是白盒测试？

尽管通常我们会拖到测试后期才考虑这个问题，但我喜欢早点做出选择。不妨跟面试官确认一下，要做黑盒测试还是白盒测试——或是两者都要。

- 步骤2：使用者是哪些人？做什么用？

一般来说，软件都会有一个或多个目标用户，设计各个功能时，就会考虑用户需求。比如，若要你测试一款家长用来监控网页浏览器的软件，那么，你的目标用户既包括家长（实施监控过滤哪些网站），又包括孩子（有些网站被过滤了）。用户也可能包括“访客”（也就是既不实施也不受监控的使用者）。

- 步骤3：有哪些用例？

在监控过滤软件中，家长的用例包括安装软件、更新过滤网站清单、移除过滤网站，以及供他们自己使用的不受限制的网络。对孩子而言，用例包括访问合法内容及“非法”内容。

切记，不可凭空想象来决定各种用例，应该与面试官交流讨论后确定。

- 步骤4：有哪些使用限制？

大致定义好用例后，我们还需找出确切的意思。“网络被过滤屏蔽”具体指什么？只过滤屏蔽“非法”网页还是屏蔽整个网站？是否要求该软件具备“学习”能力，从而识别不良内容，抑或只是根据白名单或黑名单进行过滤？若要求具备学习能力并自动识别不良内容，允许多大的误报漏报率？

- 步骤5：压力条件和失效条件为何？

软件的失效是不可避免的，那么，软件失效应该是什么样的？显然，就算软件失效了，也不能导致计算机宕机。在本例中，失效可能是软件未能屏蔽本该屏蔽的网站，或是屏蔽本来允许访问的网站。对于后一种情况，你或许应该与面试官讨论一下，是不是要让家长输入密码，允许访问该网站。

- 步骤6：有哪些测试用例？如何执行测试？

这里才是手动测试和自动测试以及黑盒测试和白盒测试真正显示出差异的地方。

在步骤3和步骤4中，我们初步拟定了软件的用例，这里会进一步加以定义，并讨论该如何执行测试。具体需要测试哪些情况？其中哪些步骤可以自动化？哪些又需要人工介入？

请记住，在有些测试中，虽然自动化可以助你一臂之力，但它也有着重大缺陷。一般来说，在测试过程中，手动测试还是少不了的。

对着上面的清单一步步解决问题时，请不要想到什么就草率吐露。这会显得很没有条理，而且你肯定会遗漏重要环节。相反，你应该组织好自己的思路，先将测试工作分割为几个主要模块，

然后逐一展开分析。这样，不仅可以给出一份更完整的测试用例清单，而且也显得你做事有层次、有条理。

4. 测试一个函数

基本上，测试函数是测试中最简单的一种，与面试官的交流相对也会比较简短、清晰，因为，测试一个函数通常不外乎就是验证输入与输出。

话说回来，千万不要忽视与面试官交流的重要性。对于任意可能，特别是如何处理特定情况，你都应该深究到底。

假设要你编写代码测试对整数数组排序的函数`sort(int[] array)`，可参考下面的解决步骤。

- 步骤1：定义测试用例

一般来说，你应该考虑以下几种测试用例。

- **正常情况**：输入正常数组时，该函数是否能生成正确的输出？务必记得考虑其中的潜在问题。比如，排序通常涉及某种分割处理，应该要合理的想一想，数组元素个数为奇数时，由于无法均分数组，算法可能无法处理。所以，测试用例必须涵盖元素个数为偶数与奇数的两种数组。
- **极端情况**：传入空数组会出现什么问题？或传入一个很小的数组（只有一个元素）？此外，传入非常大数组又会如何呢？
- **空指针和“非法”输入**：值得花时间好好考虑一番，若函数接收到非法输入，该怎么处理。比如，你在测试生成第n项斐波那契数的函数，那么，在测试用例中，自然要考虑n为负数的情况。
- **奇怪的输入**：第四种有可能出现的情况：奇怪的输入。传入一个有序数组会怎么样？或者，传入一个反向排序的数组呢？

只有充分了解函数功能，才能想到这些测试用例。如果你对各种限制条件不是很清楚，最好先向面试官问个清楚。

- 步骤2：定义预期结果

通常，预期结果非常明显：正确的输出。然而，在某些情况下，你可能还需要验证其他情况。比如，如果`sort`函数返回的是一个已排序的新数组，那么，你可能还要验证一下原先的数组是否保持原样。

- 步骤3：编写测试代码

有了测试用例，并定义好预期结果后，编写代码实现这些测试用例，也就水到渠成了。代码大致如下：

```
void testAddThreeSorted() {
   MyList list = new MyList();
   list.addThreeSorted(3, 1, 2); // 按顺序添加3个元素
   assertEquals(list.getElement(0), 1);
   assertEquals(list.getElement(1), 2);
   assertEquals(list.getElement(2), 3);
}
```

5. 调试与故障排除

测试问题的最后一种是，说明你会如何调试或排除已知故障。碰到这种问题，很多求职者都会支支吾吾，处理不当，给出诸如“重装软件”等不切实际的答案。其实，就像其他问题一样，还是有章可循的，也可以有条不紊地处理。

下面通过一个例子辅助说明，假设你是谷歌Chrome浏览器团队的一员，收到一份bug报告：Chrome启动时会崩溃。你会怎么处理？

重新安装浏览器或许就能解决该用户的问题，但是，若其他用户碰到同样问题，怎么办？你的目标是搞清楚究竟出了什么问题，以便开发人员修复缺陷。

- 步骤1：理清状况

首先，你应该多提问题，尽量了解当时的情况：

- ❑ 用户碰到这个问题有多久了？
- ❑ 该浏览器的版本号？在什么操作系统下运行？
- ❑ 该问题经常发生吗？或者，出问题的频率有多高？什么时候会发生？
- ❑ 有无提交错误报告？

- 步骤2：分解问题

了解了问题发生时的具体状况，接下来，着手将问题分解为可测模块。在这个例子中，可以设想出以下操作步骤。

(1) 转到Windows的“开始”菜单。

(2) 点击Chrome图标。

(3) 浏览器启动。

(4) 浏览器载入参数设置。

(5) 浏览器发送HTTP请求载入首页。

(6) 浏览器收到HTTP回应。

(7) 浏览器解析网页。

(8) 浏览器显示网页内容。

在上述过程中的某一点，有地方出错致使浏览器崩溃。优秀的测试人员会逐一排查每个步骤，诊断定位问题所在。

- 步骤3：创建特定的、可控的测试

以上各个测试模块都应该有实际的指令动作——也就是你要求用户执行的、或是你自己可以做的操作步骤（从而在你自己的机器上予以重现）。在真实世界中，你面对的是一般客户，不可能给他们做不到或不愿做的操作指令。

面试题目

12.1 找出以下代码中的错误（可能不止一处）：

```
unsigned int i;
for (i = 100; i >= 0; --i)
  printf( “%d\n”, i);
```

（第269页）

12.2 有个应用程序一运行就崩溃，现在你拿到了源码。在调试器中运行10次之后，你发现该应用每次崩溃的位置都不一样。这个应用只有一个线程，并且只调用C标准库函数。究竟是什么样的编程错误导致程序崩溃？该如何逐一测试每种错误？（第270页）

12.3 有个国际象棋游戏程序使用了方法：`boolean canMoveTo(int x, int y)`，这个方法是`Piece`类的一部分，可以判断某个棋子能否移动到位置(x, y)。请说明你会如何测试该方法。（第271页）

12.4 不借助任何测试工具，该如何对网页进行负载测试？（第272页）

12.5 如何测试一支笔？（第272页）

12.6 在一个分布式银行系统中，该如何测试一台ATM机（自动柜员机）？（第273页）

8.13 C 和 C++

好的面试官不会要求你用自己不懂的语言来编写代码。一般来说，如果面试官要求你用C++写代码，那么，应该是你在简历上提及了C++。要是没能记住所有API，也不用担心，大部分面试官（虽不是全部）并不会那么在意这一点。不过，我们仍建议你学会基本的C++语法，这样才能轻松应对这些问题。

1. 类和继承

虽然C++的类与其他语言的类有些特征相似，不过，还是有必要回顾一下相关部分语法。

下面的代码演示了怎样利用继承实现一个基本的类。

```
#include <iostream>
using namespace std;

#define NAME_SIZE 50 // 定义一个宏

class Person {
   int id; // 所有成员默认为私有（private）
   char name[NAME_SIZE];

  public:
   void aboutMe() {
     cout << “I am a person.”;
   }
};

class Student : public Person {
  public:
   void aboutMe() {
     cout << “I am a student.”;
   }
};

int main() {
   Student * p = new Student();
   p->aboutMe(); // 打印 “I am a student.”
```

```
    delete p; // 注意！务必释放之前分配的内存
    return 0;
}
```

在C++中，所有数据成员和方法均默认为私有（private），可用关键字public修改其属性。

2. 构造函数和析构函数

对象创建时，会自动调用类的构造函数。如果没有定义构造函数，编译器会自动生成一个默认构造函数（Default Constructor）。另外，我们也可以定义自己的构造函数。

```
Person(int a) {
   id = a;
}
```

这个类的数据成员也可以这样初始化：

```
Person(int a) : id(a) {
   ...
}
```

在真正的对象创建之前，且在构造函数余下部分代码调用前，数据成员id就会被赋值。在常量数据成员赋值时（只能赋一次值），这种写法特别适用。

析构函数会在对象删除时执行清理工作。对象销毁时，会自动调用析构函数。我们不会显式调用析构函数，因此它不能带参数。

```
~Person() {
   delete obj; // 释放之前这个类里分配的内存
}
```

3. 虚函数

在前面的例子中，我们将p定义为Student类型指针变量：

```
Student * p = new Student();
p->aboutMe();
```

像下面这样，把p定义为Person *又会怎么样？

```
Person * p = new Student();
p->aboutMe();
```

这么改的话，执行时会打印“I am a person”。这是因为函数aboutMe是在编译期决定的，也即所谓的静态绑定（static binding）机制。

若要确保调用的是Student的aboutMe函数实现，可以将Person类的aboutMe定义为virtual：

```
class Person {
   ...
   virtual void aboutMe() {
     cout << “I am a person.”;
   }
};

class Student : public Person {
 public:
   void aboutMe() {
```

```
      cout << "I am a student.";
    }
};
```

当我们无法（或不想）实现父类的某个方法时，虚函数也能派上用场。例如，设想一下，我们想让Student和Teacher继承自Person，以便实现一个共同的方法，如addCourse(string s)。不过，对Person调用addCourse方法没有多大意义，因为要看对象到底是Student还是Teacher，才能确定该调用哪个方法的具体实现。

在这种情况下，我们可能想将Person类的addCourse定义为虚函数，至于函数实现则留给子类。

```
class Person {
   int id; // 所有成员默认为私有
   char name[NAME_SIZE];
  public:
   virtual void aboutMe() {
     cout << "I am a person." << endl;
   }
   virtual bool addCourse(string s) = 0;
};

class Student : public Person {
  public:
   void aboutMe() {
     cout << "I am a student." << endl;
   }

   bool addCourse(string s) {
     cout << "Added course " << s << " to student." << endl;
     return true;
   }
};

int main() {
   Person * p = new Student();
   p->aboutMe(); // 打印"I am a student."
   p->addCourse("History");
   delete p;
}
```

注意，将addCourse定义为纯虚函数，Person就成了一个抽象类，不能实例化。

- *虚析构函数*

有了虚函数，很自然地就会出现虚析构函数的概念。假设我们想要实现Person和Student的析构函数。不假思索的话，可能会写出类似如下的代码：


```
class Person {
 public:
   ~Person() {
     cout << "Deleting a person." << endl;
   }
};
```

```

class Student : public Person {
 public:
   ~Student() {
     cout << "Deleting a student." << endl;
   }
};

int main() {
   Person * p = new Student();
   delete p; // 打印"Deleting a person."
}
```

跟之前的例子一样，由于指针p指向Person，对象销毁时自然会调用Person类的析构函数。这样就会有问题，因为Student对象的内存可能得不到释放。

要解决这个问题，只需将Person的析构函数定义为虚析构函数。

```
class Person {
 public:
   virtual ~Person() {
     cout << "Deleting a person." << endl;
   }
};

class Student : public Person {
 public:
   ~Student() {
     cout << "Deleting a student." << endl;
   }
};

int main() {
   Person * p = new Student();
   delete p;
}
```

编译执行上面的代码，打印输出如下：

```
Deleting a student.
Deleting a person.
```

4. 默认值

如下所示，函数可以指定默认值。注意，所有默认参数必须放在函数声明的右边，因为没有其他途径来指定参数是怎么排列的。

```
int func(int a, int b = 3) {
   x = a;
   y = b;
   return a + b;
}

w = func(4);
z = func(4, 5);
```

5. 操作符重载

有了操作符重载（operator overloading），原本不支持+等操作符的对象，就可以用上这些操作符。举个例子，要想把两个书架（BookShelf）并作一个，我们可以这样重载+操作符：

```
BookShelf BookShelf::operator+(BookShelf &other) { ... }
```

6. 指针和引用

指针存放有变量的地址，可直接作用于变量的所有操作，都可以作用在指针上，比如访问和修改变量。

两个指针可以彼此相等，修改其中一个指针指向的值，另一个指针指向的值也会随之改变。实际上，这两个指针指向同一地址。

```
int * p = new int;
*p = 7;
int * q = p;
*p = 8;
cout << *q; // 打印8
```

注意，指针的大小随计算机的体系结构不同而不同：在32位机器上为32位，在64位机器上为64位。请谨记这一点区别，面试官常常会要求求职者准确地回答，某个数据结构到底要占用多少空间。

- 引用

引用是既有对象的另一个名字（别名），引用本身并不占用内存。例如：

```
1 int a = 5;
2 int & b = a;
3 b = 7;
4 cout << a; // 打印7
```

在上面第2行代码中，b是a的引用；修改b，a也随之改变。

创建引用时，必须指定引用指向的内存位置。当然，也可以创建一个独立的引用，如下所示：

```
/* 分配内存，储存12，
 * 声明指向这块内存的引用b */
int & b = 12;
```

跟指针不同，引用不能为空，也不能重新赋值，指向另一块内存。

- 指针算术运算

我们经常会看到开发人员对指针执行加法操作，示例如下：

```
int * p = new int[2];
p[0] = 0;
p[1] = 1;
p++;
cout << *p; // 输出1
```

执行p++会跳过sizeof(int)个字节，因此上面的代码会输出1。如果p换作其他类型，p++就会跳过一定数目（等于该数据结构的大小）的字节。

7. 模板

模板是一种代码重用方式，不同的数据类型可以套用同一个类的代码。比如说，我们可能有

列表类的数据结构，希望可以放进不同类型的数据。下面的代码通过ShiftedList类实现这一需求。

```
template <class T>
class ShiftedList {
   T* array;
   int offset, size;
 public:
   ShiftedList(int sz) : offset(0), size(sz) {
     array = new T[size];
   }

   ~ShiftedList() {
     delete [] array;
   }

   void shiftBy(int n) {
     offset = (offset + n) % size;
   }

   T getAt(int i) {
     return array[convertIndex(i)];
   }

   void setAt(T item, int i) {
     array[convertIndex(i)] = item;
   }

private:
   int convertIndex(int i) {
     int index = (i - offset) % size;
     while (index < 0) index += size;
     return index;
   }
};

int main() {
   int size = 4;
   ShiftedList<int> * list = new ShiftedList<int>(size);
   for (int i = 0; i < size; i++) {
     list->setAt(i, i);
   }
   cout << list->getAt(0) << endl;
   cout << list->getAt(1) << endl;
   list->shiftBy(1);
   cout << list->getAt(0) << endl;
   cout << list->getAt(1) << endl;
   delete list;
}
```

面试题目

13.1 用C++写个方法，打印输入文件的最后K行。（第274页）

13.2 比较并对比散列表和STL map。散列表是怎么实现的？如果输入的数据量不大，可以选用哪些数据结构替代散列表？（第275页）
13.3 C++虚函数的工作原理是什么？（第275页）
13.4 深拷贝和浅拷贝之间有何区别？请说明两者的用法。（第276页）
13.5 C语言的关键字`volatile`有何作用？（第277页）
13.6 基类的析构函数为何要声明为`virtual`？（第278页）
13.7 编写方法，传入参数为指向`Node`结构的指针，返回传入数据结构的完整拷贝。其中，`Node`数据结构含有两个指向其他`Node`的指针。（第278页）
13.8 编写一个智能指针类。智能指针是一种数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录`SmartPointer<T*>`对象的引用计数，一旦T类型对象的引用计数为零，就会释放该对象。（第279页）
13.9 编写支持对齐分配的`malloc`和`free`函数，分配内存时，`malloc`函数返回的地址必须能被2的n次方整除。

示例

`align_malloc(1000,128)`返回的内存地址可被`128`整除，并指向一块`1000`字节大小的内存。

`aligned_free()`会释放`align_malloc`分配的内存。（第281页）
13.10 用C编写一个`my2DAlloc`函数，可分配二维数组。将`malloc`函数的调用次数降到最少，并确保可通过`arr[i][j]`访问该内存。（第282页）

参考问题：数组与字符串（#1.2）；链表（#2.7）；测试（#12.1）；Java（#14.4）；线程与锁（#16.3）。

8.14　Java

虽然本书到处都是跟Java相关的问题，不过，本章探讨的是Java及其语法方面的问题。较大的公司通常不会考这类问题，这些公司偏重于测试求职者的资质而非知识，也有时间和资源就特定语言对求职者进行培训。不过，若在其他公司，这类棘手的问题可能相当常见。

1. 如何处理

既然这些问题考的是你知道不知道，讨论这类问题的解法似乎有点可笑。毕竟，所谓的解法不就是要知道正确答案吗？

既是也不是。当然，掌握这些问题的最佳途径就是搞懂Java的里里外外。不过，若在处理问题时卡壳了，不妨试试下面的方法。

(1) 根据情况创建实例，问问自己该如何推演。

(2) 问问自己，换作其他语言，该怎么处理这种情况。

(3) 如果你是语言设计者，该怎么设计？各种设计选择都会造成什么影响？

相比不假思索地答出问题，如果你能推导出答案，同样会给面试官留下深刻的印象。不要试图蒙混过关。你可以直接告诉面试官："我不确定能否想起答案，不过让我试试能不能搞定

它。假设我们拿到这段代码……”

- 关键字final

Java语言的关键字final用于变量、类或方法时，含义各不相同。

- **变量**：一旦初始化，变量值就不能修改。
- **方法**：该方法不能被子类重写（override）。
- **类**：该类不能派生子类。

- 关键字finally

关键字finally和try/catch语句块配对使用，即使有异常抛出，也能确保某段代码一定会执行。finally语句块会在try和catch语句块之后，在控制权交回之前执行。

注意，下面这个例子中该关键字是怎么起作用的。

```
public static String lem() {
   System.out.println("lem");
   return "return from lem";
}

public static String foo() {
   int x = 0;
   int y = 5;
   try {
     System.out.println("start try");
     int b = y / x;
     System.out.println("end try");
     return "returned from try";
   } catch (Exception ex) {
     System.out.println("catch");
     return lem() + " | returned from catch";
   } finally {
     System.out.println("finally");
   }
}

public static void bar() {
   System.out.println("start bar");
   String v = foo();
   System.out.println(v);
   System.out.println("end bar");
}

public static void main(String[] args) {
   bar();
}
```

这段代码的输出如下：

```
start bar
start try
catch
lem
finally
return from lem | returned from catch
end bar
```

注意上述输出的第3 ~ 5行。整个catch语句块都会执行（包括return语句里的函数调用），然后执行finally语句块，之后该函数才真正返回。

- finalize方法

在真正销毁对象之前，自动垃圾收集器会调用finalize()方法。因此，一个类可以重写Object类的finalize()方法，以便定义在垃圾收集时的特定行为。

```
protected void finalize() throws Throwable {
   /* 关闭已打开的文件，释放资源等 */
}
```

2. 重载与重写

重载（overloading）是指两个方法的名称相同，但参数类型或个数不同。

```
public double computeArea(Circle c) { ... }
public double computeArea(Square s) { ... }
```

而重写（overriding）是指某个方法与父类的方法拥有相同的名称和函数签名。

```
public abstract class Shape {
   public void printMe() {
     System.out.println("I am a shape.");
   }
   public abstract double computeArea();
}

public class Circle extends Shape {
   private double rad = 5;
   public void printMe() {
     System.out.println("I am a circle.");
   }

   public double computeArea() {
     return rad * rad * 3.15;
   }
}

public class Ambiguous extends Shape {
   private double area = 10;
   public double computeArea() {
     return area;
   }
}

public class IntroductionOverriding {
   public static void main(String[] args) {
     Shape[] shapes = new Shape[2];
     Circle circle = new Circle();
     Ambiguous ambiguous = new Ambiguous();

     shapes[0] = circle;
     shapes[1] = ambiguous;

```

```
    for (Shape s : shapes) {
       s.printMe();
       System.out.println(s.computeArea());
    }
  }
}
```

这段代码的输出如下：

```
I am a circle.
78.75
I am a shape.
10.0
```

由此可见，Circle重写了printMe()，但Ambiguous并未重写该方法。

3. 集合框架

Java的集合框架（collection framework）极其有用，本书许多章节都用到了。下面介绍几个最常用的。

ArrayList：ArrayList是一种可动态调整大小的数组，随着元素的插入，数组会适时扩容。

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("one");
myArr.add("two");
System.out.println(myArr.get(0)); /* 打印<one> */
```

Vector：Vector与ArrayList非常类似，只不过前者是同步的（synchronized）。两者语法也相差无几。

```
Vector<String> myVect = new Vector<String>();
myVect.add("one");
myVect.add("two");
System.out.println(myVect.get(0));
```

LinkedList：这里说的LinkedList当然是Java内建的LinkedList类。LinkedList在面试中很少出现，不过值得学习研究，因为使用时会引出一些迭代器的语法。

```
LinkedList<String> myLinkedList = new LinkedList<String>();
myLinkedList.add("two");
myLinkedList.addFirst("one");
Iterator<String> iter = myLinkedList.iterator();
while (iter.hasNext()) {
   System.out.println(iter.next());
}
```

HashMap：HashMap集合广泛用于各种场合，不论是在面试中，还是在实际开发中。下面展示了HashMap的部分语法。

```
HashMap<String, String> map = new HashMap<String, String>();
map.put("one", "uno");
map.put("two", "dos");
System.out.println(map.get("one"));
```

面试之前，确保自己对上述语法了如指掌。这些语法派得上用场。

面试题目

请注意，本书几乎所有问题的解决方法都采用Java实现，因此这里只列了几个问题。而且，这些问题主要涉及Java语言的细枝末节，毕竟本书其余章节中有很多Java有关的编程问题。

14.1 从继承的角度来看，将构造函数声明为私有会有何作用？（第284页）

14.2 在Java中，若在`try-catch-finally`的`try`语句块中插入`return`语句，`finally`语句块是否还会执行？（第284页）

14.3 `final`、`finally`和`finalize`之间有何差异？（第285页）

14.4 C++模板和Java泛型之间有何不同？（第285页）

14.5 Java中的对象反射是什么？它有什么用？（第287页）

14.6 实现`CircularArray`类，支持类似数组的数据结构，这些数据结构可以高效地进行旋转。该类应该使用泛型，并通过标准的`for (Obj o : circularArray)`语法支持迭代操作。（第287页）

参考问题：数组与字符串（#1.4）；面向对象设计（#8.10）；线程与锁（#16.3）。

8.15 数据库

有数据库经验的求职者可能会被要求实现SQL查询，或是设计应用程序所需的数据库，以确认你掌握这方面的知识。本章将回顾一些关键概念，并简述如何解决这些问题。

看到这些查询时，对于语法上的细微差异，不必太惊讶。SQL的版本和变体很多，下面这些SQL与你之前接触过的可能稍有不同。本书的SQL示例已在微软SQL Server经过测试。

1. SQL语法及各类变体

开发人员常常会在SQL查询中使用隐式连接（implicit join）和显式连接（explicit join）。两者的语法如下。

```
/* 显式连接 */
SELECT CourseName, TeacherName
FROM   Courses INNER JOIN Teachers
ON     Courses.TeacherID = Teachers.TeacherID

/* 隐式连接 */
SELECT CourseName, TeacherName
FROM   Courses, Teachers
WHERE     Courses.TeacherID = Teachers.TeacherID
```

上面两条语句的作用是等价，至于选用哪条全看个人喜好。为保持前后一致，我们将一直使用显式连接。

2. 非规范化和规范化数据库

规范化数据库的设计目标是将冗余降到最低，而非规范化数据库则是为了优化读取时间。

在传统的规范化数据库中，若有诸如`Courses`和`Teachers`的数据，`Courses`可能含有`TeacherID`列，这是指向`Teachers`的外键（foreign key）。这么做的好处之一是，关于教师的信息

（姓名、住址等）在数据库中只有一份。而缺点是大量常用的查询需要执行开销很大的连接操作。

反之，我们可以存储冗余数据，使数据库非规范化。例如，若能预计到这类查询会频繁执行，可以将教师姓名存到Courses表中。非规范化通常用于构建高可扩展性系统。

3. SQL语句

下面以前面提到的数据库为例，复习一下基本的SQL语法。这个数据库的简单结构如下，其中*表示主键：

```
Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName
StudentCourses: CourseID*, StudentID*
```

根据上面这些表，实现下列查询。

- 查询1：学生选课情况

实现一个查询，列出所有学生，以及每个学生选修了几门课程。

首先，我们或许可以试着这么写：

```
/* 错误的代码 */
SELECT Students.StudentName, count(*)
FROM Students INNER JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
GROUP BY Students.StudentID
```

上述查询有以下三个问题。

(1) 我们将一门课都没选的学生排除掉了，因为StudentCourses只包括已经选课的学生。我们可以把INNER JOIN改为LEFT JOIN（左连接）。

(2) 即使改为LEFT JOIN，上面的查询还是不太对。count(*)会返回一组StudentID里有几项。一门课都没选的学生在对应的组里仍有一项。这里需要将count(*)改为计数每个组里CourseID的数量：count(StudentCourses.CourseID)。

(3) 上面的查询已按Students.StudentID分组，但每个组仍有多个StudentName。数据库怎么知道该返回哪个StudentName？当然，它们的值可能都一样，但数据库并不知道这点。这里需要运用聚合（aggregate）函数，比如first(Students.StudentName)。

修正上述问题后，得到如下查询：

```
1  /* 解法1：用另一个查询包裹起来 */
2  SELECT StudentName, Students.StudentID, Cnt
3  FROM (
4     SELECT  Students.StudentID,
5                  count(StudentCourses.CourseID) as [Cnt]
6     FROM Students LEFT JOIN StudentCourses
7     ON Students.StudentID = StudentCourses.StudentID
8     GROUP BY Students.StudentID
9  ) T INNER JOIN Students on T.studentID = Students.StudentID
```

看到这段代码，有人可能会问，为什么不直接在第3行里选出学生姓名，就不需要第3到第6行的另一个查询了。这么做的话，就会得到如下（错误的）解法：

```
/* 错误的代码 */
SELECT StudentName, Students.StudentID,
       count(StudentCourses.CourseID) as [Cnt]
FROM Students LEFT JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
GROUP BY Students.StudentID
```

答案是我们不能这么改——至少不能一五一十照上面那样改。我们只能选择聚合函数或GROUP BY子句里的值。

另外，我们可以使用下面两条语句之一解决上面的问题：

```
/* 解法2：在GROUP BY子句中加入StudentName */
SELECT StudentName, Students.StudentID,
            count(StudentCourses.CourseID) as [Cnt]
FROM Students LEFT JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
GROUP BY Students.StudentID, Students.StudentName
```

或

```
/* 解法3：用聚合函数包裹起来 */
SELECT  max(StudentName) as [StudentName], Students.StudentID,
             count(StudentCourses.CourseID) as [Count]
FROM Students LEFT JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
GROUP BY Students.StudentID
```

- 查询2：教师班级规模

实现一个查询，取得一份所有教师的列表，以及每位教师要教多少学生。如果一位教师给某个学生教授两门课程，那么，这个学生就要计入两次。根据教师教授的学生人数，将结果列表从大到小进行排序。

下面逐步构造这个查询。首先，取得一份TeacherID列表，以及有多少学生跟各个TeacherID有关联。这跟前一个查询非常相似。

```
SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
FROM Courses INNER JOIN StudentCourses
ON Courses.CourseID = StudentCourses.CourseID
GROUP BY Courses.TeacherID
```

请注意，这里的INNER JOIN不会选取那些不教课的教师。我们会在下面的查询中进行处理，将它与包含所有教师的列表相连接。

```
SELECT TeacherName, isnull(StudentSize.Number, 0)
FROM Teachers LEFT JOIN
     (SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
      FROM Courses INNER JOIN StudentCourses
      ON Courses.CourseID = StudentCourses.CourseID
      GROUP BY Courses.TeacherID) StudentSize
ON Teachers.TeacherID = StudentSize.TeacherID
ORDER BY StudentSize.Number DESC
```

请注意，上面的查询是如何在SELECT语句中处理NULL值的：将NULL值转换为零。

8

4. 小型数据库设计

另外，面试官或许会让你自己设计一个数据库。下面会逐步剖析一种设计方法。你可能会发现该方法与面向对象设计方法存在相似之处。

- 步骤1：处理不明确的地方

不管是有意还是无意，数据库问题往往存在含糊不清的地方。开始设计之前，你必须准确理解自己要设计什么。

设想一下，你被要求设计一套系统，供公寓租赁中介使用。你需要弄清楚这家中介有多栋楼还是只有一栋，而且还应该跟面试官讨论系统的通用性要做到什么程度。比如，某人租用同一栋楼里的两套公寓的情况极为少见，但这是否意味着你用不着处理这种情况？也许是，也许不是。有些非常罕见的条件或许最好做变通处理（比如，在数据库中，重复存储承租人的联系信息）。

- 步骤2：定义核心对象

接下来，该来看看系统的核心对象了。一般来说，每个核心对象都会转变为一张表。在这个例子中，核心对象可能包括Property（财产）、Building（大楼）、Apartment（公寓）、Tenant（承租人）和Manager（管理员）。

- 步骤3：分析表之间的关系

勾勒出核心对象后，我们就可以比较清晰地知道这些表该是什么样的。这些表之间有何关联呢？它们的关系是多对多？还是一对多？

若Building和Apartment有一对多的关系（一幢Building会有很多Apartment），那么，也许可以表示如下：

Buildings		
BuildingID	BuildingName	BuildingAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

注意，Apartments表通过BuildingID列链接回Buildings。

若允许承租人租用多套公寓，那么，可能就要实现多对多关系，如下所示：

Tenants		
TenantID	TenantName	TenantAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

TenantApartments	
TenantID	ApartmentID

TenantApartments表储存Tenants和Apartments之间的关系。

● 步骤4：研究该有什么操作动作

最后，我们要填充细节。想想常见的操作动作，弄清楚如何存入和取回相关数据。我们还需处理租赁条款、腾空房间、租金付款等。每个动作都需要新的表和列。

5. 大型数据库设计

在设计大型、可扩展的数据库时，上述例子用到的连接（join）通常都很慢。因此，你必须对数据做非规范化处理。请仔细想想数据会怎么使用——你可能需要在多个表中重复储存同一份数据。

面试题目

问题1 ~ 3用到以下数据库模式：

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar
BuildingID	int	BuildingName	varchar		
		Address	varchar		

Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

注意每套公寓可能有多位承租人，而每位承租人可能租住多套公寓。每套公寓只属于一栋大楼，而每栋大楼属于一个综合体。

15.1 编写SQL查询，列出租住不止一套公寓的承租人。（第290页）

15.2 编写SQL查询，列出所有建筑物，并取得状态为“Open”的申请数量（Requests表中Status为Open的条目）。（第291页）

15.3 11号建筑物正在进行大翻修。编写SQL查询，关闭这栋建筑物里所有公寓的入住申请。（第291页）

15.4 连接有哪些不同类型？请说明这些类型之间的差异，以及为何在某些情形下，某种连接会比较好。（第291页）

15.5 什么是反规范化？请说明优缺点。（第292页）

15.6 有个数据库，里面有公司（companies）、人（people）和专业人员（professionals，为公司工作），请绘制实体关系图。（第293页）

15.7 给定一个储存有学生成绩的简单数据库。设计这个数据库的大概样子，并编写SQL查询，返回优等生名单（排名前10%），以平均分排序。（第293页）

参考问题：面向对象设计（#8.6）。

8

8.16 线程与锁

在微软、谷歌或亚马逊等公司的面试中，求职者被要求以线程实现算法的情况并不是很常见（除非你打算加入的团队特别看重这方面的技能）。不过，不管是什么公司，面试官常常会考你对线程有没有一定程度的了解，特别是对死锁的理解。

本章将简要介绍这个主题。

1. Java线程

在Java中，每个线程的创建和控制都是由`java.lang.Thread`类的独特对象实现的。一个独立的应用运行时，会自动创建一个用户线程，执行`main()`方法。这个线程叫作主线程。

在Java中，实现线程有以下两种方式：

- 通过实现`java.lang.Runnable`接口；
- 通过扩展`java.lang.Thread`类。

下面介绍这两种方式。

- **实现Runnable接口**

`Runnable`接口的结构非常简单：

```
public interface Runnable {
   void run();
}
```

要用这个接口创建和使用线程，步骤如下。

(1) 创建一个实现`Runnable`接口的类，该类的对象是一个`Runnable`对象。

(2) 创建一个`Thread`类型的对象，并将`Runnable`对象作为参数传入`Thread`构造函数。于是，这个`Thread`对象包含一个实现`run()`方法的`Runnable`对象。

(3) 调用上一步创建的`Thread`对象的`start()`方法。

示例如下：

```
public class RunnableThreadExample implements Runnable {
   public int count = 0;

   public void run() {
     System.out.println("RunnableThread starting.");
     try {
       while (count < 5) {
         Thread.sleep(500);
         count++;
       }
     } catch (InterruptedException exc) {
        System.out.println("RunnableThread interrupted.");
     }
     System.out.println("RunnableThread terminating.");
   }
}

```

```
18 public static void main(String[] args) {
19    RunnableThreadExample instance = new RunnableThreadExample();
20    Thread thread = new Thread(instance);
21    thread.start();
22
23    /* 等到上面的线程数到5（时间有点长） */
24    while (instance.count != 5) {
25      try {
26         Thread.sleep(250);
27      } catch (InterruptedException exc) {
28         exc.printStackTrace();
29      }
30    }
31 }
```

从上面的代码可以看出，我们真正需要做的是我们的类必须实现`run()`方法（第4行）。然后，另一个方法就可以将这个类的实例传入`new Thread(obj)`（第19～20行），然后调用那个线程的`start()`（第21行）。

- 扩展Thread类

创建线程还有一种方式，就是通过扩展`Thread`类实现。使用这种方式，基本上就意味着要重写`run()`方法，并且在子类的构造函数里，还需要显式调用这个线程的构造函数。

下面是使用这种方式的示例代码。

```
public class ThreadExample extends Thread {
   int count = 0;

   public void run() {
     System.out.println("Thread starting.");
     try {
       while (count < 5) {
         Thread.sleep(500);
         System.out.println("In Thread, count is " + count);
        count++;
       }
     } catch (InterruptedException exc) {
        System.out.println("Thread interrupted.");
     }
     System.out.println("Thread terminating.");
   }
}

public class ExampleB {
   public static void main(String args[]) {
     ThreadExample instance = new ThreadExample();
     instance.start();

     while (instance.count != 5) {
        try {
          Thread.sleep(250);
        } catch (InterruptedException exc) {
          exc.printStackTrace();
```

```
        }
      }
    }
}
```

这段代码跟之前的做法非常相似。两者的区别在于，既然我们是扩展Thread类而非只是实现一个接口，因此可以在这个类的实例中调用start()。

- 扩展Thread类 vs. 实现Runnable接口

在创建线程时，相比扩展Thread类，实现Runnable接口可能更优，理由有二。

- ❑ Java不支持多重继承。因此，扩展Thread类也就代表这个子类不能扩展其他类。而实现Runnable接口的类还能扩展另一个类。
- ❑ 类可能只要求可执行即可，因此，继承整个Thread类的开销过大。

2. 同步和锁

给定一个进程内的所有线程，都共享同一存储空间，这样有好处又有坏处。这些线程就可以共享数据，非常有用。不过，在两个线程同时修改某一资源时，这也会造成一些问题。Java提供了同步机制，以控制对共享资源的访问。

关键字synchronized和lock构成了代码同步执行的实现基础。

- 同步方法

最常见的做法是，使用关键字synchronized对共享资源的访问加以限制。该关键字可以用在方法和代码块上，限制多个线程，使之不能同时执行**同一个对象**的代码。

要搞清楚最后一点，请看以下代码：

```
public class MyClass extends Thread  {
   private String name;
   private MyObject myObj;

   public MyClass(MyObject obj, String n) {
     name = n;
     myObj = obj;
   }

   public void run() {
     myObj.foo(name);
   }
}

public class MyObject {
   public synchronized void foo(String name) {
     try {
        System.out.println("Thread " + name + ".foo(): starting");
        Thread.sleep(3000);
        System.out.println("Thread " + name + ".foo(): ending");
     } catch (InterruptedException exc) {
        System.out.println("Thread " + name + ": interrupted.");
     }
   }
}
```

若有两个MyClass实例，能否同时调用foo？这要看情况，若它们共用一个MyObject实例，则答案是不可以。但是，若两个实例持有不同的引用，那么，答案就是可以。

```
/* 不同的引用——两个线程都能调用MyObject.foo() */
MyObject obj1 = new MyObject();
MyObject obj2 = new MyObject();
MyClass thread1 = new MyClass(obj1, “1”);
MyClass thread2 = new MyClass(obj2, “2”);
thread1.start();
thread2.start()

/* 相同的obj引用。只有一个线程可以调用foo，另一个线程必须等待 */
MyObject obj = new MyObject();
MyClass thread1 = new MyClass(obj, “1”);
MyClass thread2 = new MyClass(obj, “2”);
thread1.start()
thread2.start()
```

静态方法会以类锁（class lock）进行同步。上面两个线程无法同时执行同一个类的同步静态方法，即使其中一个线程调用foo而另一个线程调用bar也不行。

```
public class MyClass extends Thread  {
   ...
   public void run() {
     if (name.equals(“1”)) MyObject.foo(name);
     else if (name.equals(“2”)) MyObject.bar(name);
   }
}

public class MyObject {
   public static synchronized void foo(String name) {
     /* 同之前的foo实现 */
   }

   public static synchronized void bar(String name) {
     /* 同上面的foo方法 */
   }
}
```

执行这段代码，打印输出如下：

```
Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

- 同步块

同样，代码块也可以同步化。其操作与同步方法非常相似。

```
public class MyClass extends Thread  {
   ...
   public void run() {
     myObj.foo(name);
   }
```

```
}
public class MyObject {
   public void foo(String name) {
     synchronized(this) {
        ...
     }
   }
}
```

和同步方法一样，每个MyObject实例只有一个线程可以执行同步块中的代码。这就意味着，若thread1和thread2持有同一个MyObject实例，那么，每次只有一个线程允许执行那个代码块。

- 锁

若要实现更细粒度的控制，我们可以使用锁（lock）。锁（或监视器）用于对共享资源的同步访问，方法是将锁与共享资源关联在一起。线程必须先取得与资源关联的锁，才能访问共享资源。不管在任意时间点，最多只有一个线程能拿到锁，因此，只有一个线程可以访问共享资源。

锁的常见用法是，从多个地方访问同一资源时，同一时刻只有一个线程才能访问。以下面的代码为示范。

```
public class LockedATM {
   private Lock lock;
   private int balance = 100;

   public LockedATM() {
     lock = new ReentrantLock();
   }

   public int withdraw(int value) {
     lock.lock();
     int temp = balance;
     try {
        Thread.sleep(100);
        temp = temp - value;
        Thread.sleep(100);
        balance = temp;
     } catch (InterruptedException e) {     }
     lock.unlock();
     return temp;
   }

   public int deposit(int value) {
     lock.lock();
     int temp = balance;
     try {
        Thread.sleep(100);
        temp = temp + value;
        Thread.sleep(300);
        balance = temp;
     } catch (InterruptedException e) {     }
```

```
    lock.unlock();
    return temp;
  }
}
```

当然，上述代码做了特别处理，有意降低了withdraw（提款）和deposit（存款）的执行速度，以便演示可能会出现的问题。在实际开发中，我们不必写这种代码，但它反映的情况却非常真实。使用锁有助于保护共享资源，使其免遭篡改。

3. 死锁及死锁的预防

死锁（deadlock）是这样一种情形：第一个线程在等待第二个线程持有的某个对象锁，而第二个线程又在等待第一个线程持有的对象锁（或是由两个以上线程形成的类似情形）。由于每个线程都在等其他线程释放锁，以致每个线程都会一直这么等下去。于是，这些线程就陷入了所谓的死锁。

死锁的出现必须同时满足以下四个条件。

(1) **互斥**：某一时刻只有一个进程能访问某一资源。（或者，更准确地说，对某一资源的访问有限制。若资源数量有限，也可能出现死锁。）

(2) **持有并等待**：已持有某一资源的进程不必释放当前拥有的资源，就能要求更多的资源。

(3) **没有抢占**：一个进程不能强制另一个进程释放资源。

(4) **循环等待**：两个或两个以上的进程形成循环链，每个进程都在等待循环链中另一进程持有的资源。

若要预防死锁，只需避免上述任一条件，但这很棘手，因为其中有些条件很难满足。比如，想要避免条件1就很困难，因为许多资源同一时刻只能被一个进程使用（如打印机）。大部分预防死锁的算法都把重心放在避免条件4即循环等待上。

面试题目

16.1 线程和进程有何区别？（第296页）

16.2 如何测量上下文切换时间？（第296页）

16.3 在著名的哲学家就餐问题中，一群哲学家围坐在圆桌周围，每两位哲学家之间有一根筷子。每位哲学家需要两根筷子才能用餐，并且一定会先拿起左手边的筷子，然后才会去拿右手边的筷子。如果所有哲学家在同一时间拿起左手边的筷子，就有可能造成死锁。请使用线程和锁，编写代码模拟哲学家就餐问题，避免出现死锁。（第298页）

16.4 设计一个类，只有在不可能发生死锁的情况下，才会提供锁。（第299页）

16.5 给定以下代码：

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

同一个Foo实例会被传入3个不同的线程。threadA会调用first，threadB会调用second，threadC会调用third。设计一种机制，确保first会在second之前调用，second会在third之前调用。（第304页）

16.6 给定一个类，内含同步方法A和普通方法B。在同一个程序实例中，有两个线程，能否同时执行A？两者能否同时执行A和B？（第305页）

8.17　中等难题

17.1 编写一个函数，不用临时变量，直接交换两个数。（第306页）

17.2 设计一个算法，判断玩家是否赢了井字游戏。（第307页）

17.3 设计一个算法，算出n阶乘有多少个尾随零。（第310页）

17.4 编写一个方法，找出两个数字中最大的那一个。不得使用if-else或其他比较运算符。（第311页）

17.5 珠玑妙算游戏（The Game of Master Mind）的玩法如下。

计算机有四个槽，每个槽放一个球，颜色可能是红色（R）、黄色（Y）、绿色（G）或蓝色（B）。例如，计算机可能有RGGB四种（槽1为红色，槽2、3为绿色，槽4为蓝色）。

作为用户，你试图猜出颜色组合。打个比方，你可能会猜YRGB。

要是猜对某个槽的颜色，则算一次“猜中”；要是只猜对颜色但槽位猜错了，则算一次“伪猜中”。注意，“猜中”不能算入“伪猜中”。

举个例子，实际颜色组合为RGBY，而你猜的是GGRR，则算一次猜中，一次伪猜中。

给定一个猜测和一种颜色组合，编写一个方法，返回猜中和伪猜中的次数。（第313页）

17.6 给定一个整数数组，编写一个函数，找出索引m和n，只要将m和n之间的元素排好序，整个数组就是有序的。注意：$n-m$越小越好，也就是说，找出符合条件的最短序列。

示例：

输入：1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19

输出：(3, 9)（第314页）

17.7 给定一个整数，打印该整数的英文描述（例如“One Thousand, Two Hundred Thirty Four”）。（第316页）

17.8 给定一个整数数组（有正数有负数），找出总和最大的连续数列，并返回总和。

示例：

输入：2, -8, 3, -2, 4, -10

输出：5（即{3, -2, 4}）（第318页）

17.9 设计一个方法，找出任意指定单词在一本书中的出现频率。（第319页）

17.10 XML非常冗长，你找到一种编码方式，可将每个标签对应为预先定义好的整数值，该编码方式的语法如下：

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> 映射至某个预定义的整数值
Value     --> 字符串值 END
```

例如，下列XML会被转换压缩成下面的字符串（假定对应关系为family -> 1、person -> 2、firstName -> 3、lastName -> 4、state -> 5）。

```
<family lastName="McDowell" state="CA">
   <person firstName="Gayle">Some Message</person>
</family>
```

变为：

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0.
```

编写代码，打印XML元素编码后的版本（传入Element和Attribute对象）。（第320页）

17.11 给定rand5()，实现一个方法rand7()。也即，给定一个产生0到4（含）随机数的方法，编写一个产生0到6（含）随机数的方法。（第321页）

17.12 设计一个算法，找出数组中两数之和为指定值的所有整数对。（第323页）

17.13 有个简单的类似结点的数据结构BiNode，包含两个指向其他结点的指针：

```
public class BiNode {
   public BiNode node1, node2;
   public int data;
}
```

数据结构BiNode可用来表示二叉树（其中node1为左子结点，node2为右子结点）或双向链表（其中node1为前趋结点，node2为后继结点）。编写一个方法，将二叉查找树（用BiNode实现）转换为双向链表。要求所有数值的排序不变，转换操作不得引入其他数据结构（即直接操作原先的数据结构）（第324页）

17.14 哦，不！你刚刚写好一篇长文，却倒霉地误用了“查找/替换”，不慎删除了文档中所有空格、标点，大写变成小写。比如，句子“I reset the computer. It still didn’t boot!”（我重启了电脑，但还没启动好！）变成了“iresetthecomputeritstilldidntboot”。你发现，只要能正确分离各个单词，加标点和调整大小写都不成问题。大部分单词在字典里都找得到，有些字符串如专有名词则找不到。

给定一个字典（一组单词），设计一个算法，找出拆分一连串单词的最佳方式。这里“最佳”的定义是，解析后无法辨识的字符序列越少越好。

举个例子，字符串“jesslookedjustliketimherbrother”的最佳解析结果为“JESS looked just like TIM her brother”，总共有7个字符无法辨别，全部显示为大写，以示区别。（第327页）

8.18 高难度题

18.1 编写一个函数，将两个数字相加。不得使用+或其他算术运算符。（第331页）

18.2 编写一个方法，洗一副牌。要求做到完美洗牌，换言之，这副牌52!种排列组合出现的概率相同。假设给定一个完美的随机数发生器。（第332页）

18.3 编写一个方法，从大小为n的数组中随机选出m个整数。要求每个元素被选中的概率相同。（第333页）

18.4 编写一个方法，数出0到n（含）中数字2出现了几次。

示例：

输入：`25`

输出：`9（2, 12, 20, 21, 22, 23, 24和25。注意22有两个2。）`（第334页）

18.5 有个内含单词的超大文本文件，给定任意两个单词，找出在这个文件中这两个单词的最短距离（也即相隔几个单词）。有办法在$O(1)$时间里完成搜索操作吗？解法的空间复杂度如何？（第337页）

18.6 设计一个算法，给定10亿个数字，找出最小的100万个数字。假定计算机内存足以容纳全部10亿个数字。（第338页）

18.7 给定一组单词，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。

示例：

输入：`cat, banana, dog, nana, walk, walker, dogwalker`

输出：`dogwalker`（第339页）

18.8 给定一个字符串`s`和一个包含较短字符串的数组`T`，设计一个方法，根据`T`中的每一个较短字符串，对`s`进行搜索。（第341页）

18.9 随机生成一些数字并传入某个方法。编写一个程序，每当收到新数字时，找出并记录中位数。（第342页）

18.10 给定两个字典里的单词，长度相等。编写一个方法，将一个单词变换成另一个单词，一次只改动一个字母。在变换过程中，每一步得到的新单词都必须是字典里存在的。

示例：

输入：`DAMP, LIKE`

输出：`DAMP -> LAMP -> LIMP -> LIME -> LIKE`（第343页）

18.11 给定一个方阵，其中每个单元（像素）非黑即白。设计一个算法，找出四条边皆为黑色像素的最大子方阵。（第345页）

18.12 给定一个正整数和负整数组成的$N \times N$矩阵，编写代码找出元素总和最大的子矩阵。（第348页）

18.13 给定一份几百万个单词的清单，设计一个算法，创建由字母组成的最大矩形，其中每一行组成一个单词（自左向右），每一列也组成一个单词（自上而下）。不要求这些单词在清单里连续出现，但要求所有行等长，所有列等高。（第352页）

第 9 章

解题技巧

请登录我们的网站www.CrackingTheCodingInterview.com，下载完整且可编译的Java/Eclipse工程，并与其他读者一起讨论书中的面试题，提交问题，查看本书勘误，发布简历及寻求其他建议。

数据结构

- 数组与字符串
- 链表
- 栈与队列
- 树与图

概念与算法

- 位操作
- 智力题
- 数学与概率
- 面向对象设计
- 递归和动态规划
- 扩展性与存储限制
- 排序与查找
- 测试

知识类问题

- C和C++
- Java
- 数据库
- 线程与锁

附加面试题

- 中等难题
- 高难度题

9.1　数组与字符串

1.1　实现一个算法，确定一个字符串的所有字符是否全都不同。假使不允许使用额外的数据结构，又该如何处理？（第 46 页）

解法

一开始，不妨先问问面试官，上面的字符串是ASCII字符串还是Unicode字符串。这很重要，问这个问题表明你关注细节，并且对计算机科学有深刻了解。

为了简单起见，这里假定字符集为ASCII。若不是的话，则需扩大存储空间，不过其余逻辑没有分别。

假定字符集为ASCII，对于这个问题，我们可以做一个简单的优化，若字符串的长度大于字母表中的字符个数，则直接返回`false`。毕竟，若字母表只有256个字符，字符串里就不可能有280个各不相同的字符。

第一种解法是构建一个布尔值的数组，索引值`i`对应的标记指示该字符串是否含有字母表第`i`个字符。若这个字符第二次出现，则立即返回`false`。

下面是这个算法的实现代码。

```
public boolean isUniqueChars2(String str) {
   if (str.length() > 256) return false;

   boolean[] char_set = new boolean[256];
   for (int i = 0; i < str.length(); i++) {
     int val = str.charAt(i);
     if (char_set[val]) { // 这个字符已在字符串中出现过
       return false;
     }
     char_set[val] = true;
   }
   return true;
}
```

这段代码的时间复杂度为$O(n)$，其中n为字符串长度。空间复杂度为$O(1)$。

使用位向量（bit vector），可以将空间占用减少为原先的1/8。下面的代码假定字符串只含有小写字母a到z。这样一来，我们只需使用一个`int`型变量。

```
public boolean isUniqueChars(String str) {
   if (str.length() > 26) return false;

   int checker = 0;
   for (int i = 0; i < str.length(); i++) {
     int val = str.charAt(i) - ‘a’;
     if ((checker & (1 << val)) > 0) {
       return false;
     }
     checker |= (1 << val);
   }
```

```
    return true;
}
```

另外，还有以下两种解法。

(1) 将字符串中的每一个字符与其余字符进行比较。这种方法的时间复杂度为$O(n^2)$，空间复杂度为$O(1)$。

(2) 若允许修改输入字符串，可以在$O(n \log(n))$时间里对字符串排序，然后线性检查其中有无相邻字符完全相同的情况。不过，值得注意的是，很多排序算法会占用额外的空间。

从某些方面来看，这些算法算不上最优，不过，从问题的限制条件来看，或许还算是不错的解法。

1.2 用C或C++实现`void reverse(char* str)`函数，即反转一个`null`结尾的字符串。（第 46 页）

解法

这是很经典的面试题，你可能会忽略的是：不分配额外空间，直接就地反转字符串，另外，还要注意`null`字符。

下面用C语言实现整个算法。

```
void reverse(char *str) {
   char* end = str;
   char tmp;
   if (str) {
     while (*end) { /* 找出字符串末尾 */
       ++end;
     }
     --end; /* 回退一个字符，最后一个为null字符 */

     /* 从字符串首尾开始交换两个字符， */
      * 直至两个指针在中间碰头 */
     while (str < end) {
        tmp = *str;
        *str++ = *end;
        *end-- = tmp;
     }
   }
}
```

上面的代码只是实现这个解法的诸多方法之一。我们甚至还可以递归实现这段代码，但并不推荐这么做。

1.3 给定两个字符串，请编写程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。（第 46 页）

解法

跟其他许多问题一样，首先我们应该向面试官确认一些细节，弄清楚变位词（anagram）[①]比

① 变位词是由变换某个词或短语的字母顺序而构成的新的词或短语。——译者注

较是否区分大小写。比如，God是否为dog的变位词？此外，我们还应该问清楚是否要考虑空白字符。

这里假定变位词比较区分大小写，空白也要考虑在内。也就是说，“god　”不是“dog”的变位词。

比较两个字符串时，只要两者长度不同，就不可能是变位词。

解决这个问题有两个简单的解决方法，并且都采用了上述优化，即先比较字符串长度。

解法 1：排序字符串

若两个字符串互为变位词，那么它们拥有同一组字符，只不过顺序不同。因此，对字符串排序，组成这两个变位词的字符就会有相同的顺序。我们只需比较排序后的字符串。

```
public String sort(String s) {
   char[] content = s.toCharArray();
   java.util.Arrays.sort(content);
   return new String(content);
}

public boolean permutation(String s, String t) {
   if (s.length() != t.length()) {
     return false;
   }
   return sort(s).equals(sort(t));
}
```

在某种程度上，这个算法算不上最优，不过换个角度看，该算法或许更可取：它清晰、简单且易懂。从实践角度来看，这可能是解决该问题的上佳之选。

不过，要是效率当头，我们可以换种做法。

解法 2：检查两个字符串的各字符数是否相同

我们还可以充分利用变位词的定义——组成两个单词的字符数相同——来实现这个算法。我们只需遍历字母表，计算每个字符出现的次数。然后，比较这两个数组即可。

```
public boolean permutation(String s, String t) {
   if (s.length() != t.length()) {
     return false;
   }

   int[] letters = new int[256]; // 假设条件

   char[] s_array = s.toCharArray();
   for (char c : s_array) { // 计算字符串s中每个字符出现的次数
     letters[c]++;
   }

   for (int i = 0; i < t.length(); i++) {
     int c = (int) t.charAt(i);
     if (--letters[c] < 0) {
        return false;
```

```
17     }
18   }
19
20   return true;
21 }
```

注意第6行的假设条件。在面试中，最好跟面试官核实一下字符集的大小。这里假设字符集为ASCII。

1.4 编写一个方法，将字符串中的空格全部替换为“%20”。假定该字符串尾部有足够的空间存放新增字符，并且知道字符串的“真实”长度。（注：用Java实现的话，请使用字符数组实现，以便直接在数组上操作。）（第 46 页）

解法

处理字符串操作问题时，常见做法是从字符串尾部开始编辑，从后往前反向操作。这种做法很有用，因为字符串尾部有额外的缓冲，可以直接修改，不必担心会覆写原有数据。

我们将采用上面这种做法。该算法会进行两次扫描。第一次扫描先数出字符串中有多少空格，从而算出最终的字符串有多长。第二次扫描才真正开始反向编辑字符串。检测到空格则将`%20`复制到下一个位置，若不是空白，就复制原先的字符。

下面是这个算法的实现代码。

```
public void replaceSpaces(char[] str, int length) {
   int spaceCount = 0, newLength, i;
   for (i = 0; i < length; i++) {
     if (str[i] == ‘ ’) {
       spaceCount++;
     }
   }
   newLength = length + spaceCount * 2;
   str[newLength] = ‘\0’;
   for (i = length - 1; i >= 0; i--) {
     if (str[i] == ‘ ’) {
        str[newLength - 1] = ‘0’;
        str[newLength - 2] = ‘2’;
        str[newLength - 3] = ‘%’;
        newLength = newLength - 3;
     } else {
        str[newLength - 1] = str[i];
        newLength = newLength - 1;
     }
   }
}
```

因为Java字符串是不可变的（immutable），所以我们选用了字符数组来解决这个问题。若直接使用字符串，返回时就要把字符串复制一份，不过，这么做的好处是只需扫描一次。

1.5　利用字符重复出现的次数，编写一个方法，实现基本的字符串压缩功能。比如，字符串aabcccccaaa会变为a2b1c5a3。若“压缩”后的字符串没有变短，则返回原先的字符串。（第46页）

解法

乍一看，编写这个方法似乎相当简单，实则有点复杂。我们会迭代访问字符串，将字符拷贝至新字符串，并数出重复字符。这能有多难呢？

```
public String compressBad(String str) {
   String mystr = “”;
   char last = str.charAt(0);
   int count = 1;
   for (int i = 1; i < str.length(); i++) {
     if (str.charAt(i) == last) { // 找到重复字符
        count++;
     } else { // 插入字符的数目，更新last字符
        mystr += last + “” + count;
        last = str.charAt(i);
        count = 1;
     }
   }
   return mystr + last + count;
}
```

这段代码并未处理压缩后字符串比原始字符串长的情况，但除此之外，全都满足要求。这种做法效率够高吗？不妨分析一下这段代码的执行时间。

这段代码的执行时间为$O(p + k^2)$，其中p为原始字符串长度，k为字符序列的数量。比如，若字符串为aabccdeeaa，则总计有6个字符序列。执行速度慢的原因是字符串拼接操作的时间复杂度为$O(n^2)$（参见8.1节的StringBuffer部分）。

我们可以使用StringBuffer优化部分性能。

```
String compressBetter(String str) {
   /* 检查压缩后的字符串是否会变得更长 */
   int size = countCompression(str);
   if (size >= str.length()) {
     return str;
   }

   StringBuffer mystr = new StringBuffer();
   char last = str.charAt(0);
   int count = 1;
   for (int i = 1; i < str.length(); i++) {
     if (str.charAt(i) == last) { // 找到重复字符
        count++;
     } else { // 插入字符的数目，更新last字符
        mystr.append(last); // 插入字符
        mystr.append(count); // 插入数目
        last = str.charAt(i);
        count = 1;
     }
```

```
20     }
21 
22     /* 在上面第15到16行，当重复字符改变时，
23      * 才会插入字符。我们还需在函数末尾更新
24      * 字符串，因为最后一组重复字符还未放入
25      * 压缩字符串中。
26      */
27     mystr.append(last);
28     mystr.append(count);
29     return mystr.toString();
30 }
31 
32 int countCompression(String str) {
33     if (str == null || str.isEmpty()) return 0;
34     char last = str.charAt(0);
35     int size = 0;
36     int count = 1;
37     for (int i = 1; i < str.length(); i++) {
38       if (str.charAt(i) == last) {
39          count++;
40       } else {
41          last = str.charAt(i);
42          size += 1 + String.valueOf(count).length();
43          count = 1;
44       }
45     }
46     size += 1 + String.valueOf(count).length();
47     return size;
48 }
```

这个算法要好得多。注意，我们在第2～5行代码中加入了长度检查。

若不想或不能使用`StringBuffer`，我们还是可以高效地解决这个问题。第2行代码会算出字符串压缩后的长度，这样就可以构建出相应大小的字符数组，代码实现如下：

```
1  String compressAlternate(String str) {
2     /* 检查压缩后的字符串是否会变得更长 */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5       return str;
6     }
7  
8     char[] array = new char[size];
9     int index = 0;
10    char last = str.charAt(0);
11    int count = 1;
12    for (int i = 1; i < str.length(); i++) {
13      if (str.charAt(i) == last) { // 找到重复字符
14         count++;
15      } else {
16         /* 更新重复字符的数目 */
17         index = setChar(array, last, index, count);
18         last = str.charAt(i);
19         count = 1;
```

```
    }
  }

  /* 以最后一组重复字符更新字符串 */
  index = setChar(array, last, index, count);
  return String.valueOf(array);
}

int setChar(char[] array, char c, int index, int count) {
  array[index] = c;
  index++;

  /* 将数目转换成字符串，然后转成字符数组 */
  char[] cnt = String.valueOf(count).toCharArray();

  /* 从最大的数字到最小的，复制字符 */
  for (char x : cnt) {
    array[index] = x;
    index++;
  }
  return index;
}

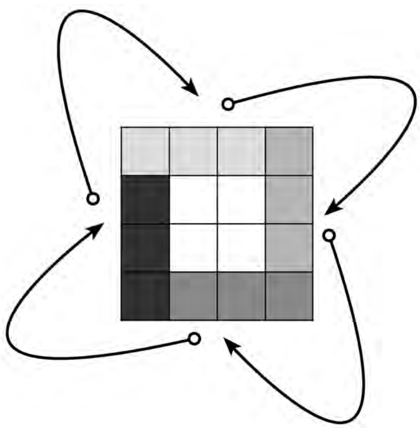
int countCompression(String str) {
  /* 与之前实现相同 */
}
```

跟第二种解法一样，执行上述代码的时间复杂度为$O(N)$，空间复杂度为$O(N)$。

1.6　给定一幅由$N \times N$矩阵表示的图像，其中每个像素的大小为 4 字节，编写一个方法，将图像旋转 90 度。不占用额外内存空间能否做到？（第 47 页）

解法

要将矩阵旋转90度，最简单的做法就是一层一层进行旋转。对每一层执行环状旋转（circular rotation），将上边移到右边、右边移到下边、下边移到左边、左边移到上边。



那么，该如何交换这四条边？一种做法是把上面复制到一个数组中，然后将左边移到上边、下边移到左边，等等。这需要占用$O(N)$内存空间，实际上没有必要。

更好的做法是按索引一个一个进行交换，具体做法如下：

```
for i = 0 to n
   temp = top[i];
   top[i] = left[i]
   left[i] = bottom[i]
   bottom[i] = right[i]
   right[i] = temp
```

从最外面一层开始逐渐向里，在每一层上执行上述交换。（另外，也可以从内层开始，逐层向外。）

下面是该算法的实现代码。

```
public void rotate(int[][] matrix, int n) {
   for (int layer = 0; layer < n / 2; ++layer) {
     int first = layer;
     int last = n - 1 - layer;
     for(int i = first; i < last; ++i) {
        int offset = i - first;
        // 存储上边
        int top = matrix[first][i];

        // 左到上
        matrix[first][i] = matrix[last-offset][first];

        // 下到左
        matrix[last-offset][first] = matrix[last][last - offset];

        // 右到下
        matrix[last][last - offset] = matrix[i][last];

        // 上到右
        matrix[i][last] = top;
     }
   }
}
```

这个算法的时间复杂度为$O(N^2)$，这已是最优的做法，因为任何算法都需要访问所有N^2个元素。

1.7 编写一个算法，若$M \times N$矩阵中某个元素为 0，则将其所在的行与列清零。（第 47 页）

解法

乍一看，这个问题似乎很简单：直接遍历整个矩阵，只要发现值为零的元素，就将其所在的行与列清零。不过这种方法有个陷阱：在读取被清零的行或列时，读到的尽是零，于是所在行与列都得变成零。很快，整个矩阵的所有元素都会变为零。

避开这个陷阱的方法之一，是新建一个矩阵标记零元素位置。然后，在第二次遍历矩阵时将零元素所在行与列清零。这种做法的空间复杂度为$O(MN)$。

真的需要占用$O(MN)$空间吗？不是的。既然打算将整行和整列清为零，因此并不需要准确记录它是cell[2][4]（行2、列4），只需知道行2有个元素为零，列4有个元素为零。不管怎样，整行和整列都要清为零，又何必要记录零元素的确切位置？

下面是这个算法的实现代码。这里用两个数组分别记录包含零元素的所有行和列。然后，在第二次遍历矩阵时，若所在行或列标记为零，则将元素清为零。

```
public void setZeros(int[][] matrix) {
   boolean[] row = new boolean[matrix.length];
   boolean[] column = new boolean[matrix[0].length];

   // 记录值为0的元素所在行索引和列索引
   for (int i = 0; i < matrix.length; i++) {
     for (int j = 0; j < matrix[0].length;j++) {
        if (matrix[i][j] == 0) {
           row[i] = true;
           column[j] = true;
        }
     }
   }

   // 若行i或列j有个元素为0，则将arr[i][j]置为0
   for (int i = 0; i < matrix.length; i++) {
     for (int j = 0; j < matrix[0].length; j++) {
        if (row[i] || column[j]) {
           matrix[i][j] = 0;
        }
     }
   }
}
```

为了提高空间利用率，我们可以选用位向量替代布尔数组。

1.8 假定有一个方法isSubstring，可检查一个单词是否为其他字符串的子串。给定两个字符串s1 和s2，请编写代码检查s2 是否为s1 旋转而成，要求只能调用一次isSubstring。（比如，waterbottle是erbottlewat旋转后的字符串。）（第 47 页）

解法

假定s2由s1旋转而成，那么，我们可以找出旋转点在哪。例如，若以wat对waterbottle旋转，就会得到erbottlewat。在旋转字符串时，我们会把s1切分为两部分：x和y，并将它们重新组合成s2。

```
s1 = xy = waterbottle
x = wat
y = erbottle
s2 = yx = erbottlewat
```

因此，我们需要确认有没有办法将s1切分为x和y，以满足xy = s1和yx = s2。不论x和y之间的分割点在何处，我们会发现yx肯定是xyxy的子串。也即，s2总是s1s1的子串。

上述分析正是这个问题的解法：直接调用isSubstring(s1s1, s2)即可。

下面是上述算法的实现代码。

```
public boolean isRotation(String s1, String s2) {
   int len = s1.length();
   /* 检查s1和s2是否等长且不为空 */
   if (len == s2.length() && len > 0) {
     /* 拼接s1和s1，放入新字符串中 */
     String s1s1 = s1 + s1;
     return isSubstring(s1s1, s2);
   }
   return false;
}
```

9.2　链表

2.1　编写代码，移除未排序链表中的重复结点。

进阶

如果不得使用临时缓冲区，该怎么解决？（第48页）

解法

要想移除链表中的重复结点，我们需要设法记录有哪些是重复的。这里只要用到一个简单的散列表。

在下面的解法中，我们会直接迭代访问整个链表，将每个结点加入散列表。若发现有重复元素，则将该结点从链表中移除，然后继续迭代。这个题目使用了链表，因此只需扫描一次就能搞定。

```
public static void deleteDups(LinkedListNode n) {
   Hashtable table = new Hashtable();
   LinkedListNode previous = null;
   while (n != null) {
     if (table.containsKey(n.data)) {
       previous.next = n.next;
     } else {
       table.put(n.data, true);
       previous = n;
     }
     n = n.next;
   }
}
```

上述代码的时间复杂度为$O(N)$，其中N为链表结点数目。

进阶：不得使用缓冲区

如不借助额外的缓冲区，可以用两个指针来迭代：current迭代访问整个链表，runner用于检查后续的结点是否重复。

```
public static void deleteDups(LinkedListNode head) {
   if (head == null) return;

   LinkedListNode current = head;
   while (current != null) {
     /* 移除后续值相同的所有结点 */
     LinkedListNode runner = current;
     while (runner.next != null) {
       if (runner.next.data == current.data) {
         runner.next = runner.next.next;
       } else {
         runner = runner.next;
       }
     }
     current = current.next;
   }
}
```

这段代码的空间复杂度为$O(1)$，但时间复杂度为$O(N^2)$。

2.2　实现一个算法，找出单向链表中倒数第*k*个结点。（第 48 页）

解法

下面会以递归和非递归的方式解决这个问题。一般来说，递归解法更简洁，但效率比较差。例如，就这个问题来说，递归解法的代码量大概只有迭代解法的一半，但要占用$O(n)$空间，其中n为链表结点个数。

注意，在下面的解法中，k定义如下：传入`k = 1`将返回最后一个结点，`k = 2`返回倒数第2个结点，依此类推。当然，也可以将k定义为`k = 0`返回最后一个结点。

解法 1：链表长度已知

若链表长度已知，那么，倒数第k个结点就是第`(length - k)`个结点。直接迭代访问链表就能找到这个结点。不过，这个解法太过简单了，不大可能是面试官想要的答案。

解法 2：递归

这个算法会递归访问整个链表，当抵达链表末端时，该方法会回传一个置为0的计数器。之后的每次调用都会将这个计数器加1。当计数器等于k时，表示我们访问的是链表倒数第k个元素。

实现代码简洁明了，前提是我们要有办法通过栈“回传”一个整数值。可惜，我们无法用一般的返回语句回传一个结点和一个计数器，那该怎么办？

- *方法A：不返回该元素*

一种方法是对这个问题略作调整，只打印倒数第k个结点的值。然后，直接通过返回值传回计数器值。

```
public static int nthToLast(LinkedListNode head, int k) {
   if (head == null) {
     return 0;
```

```
  }
  int i = nthToLast(head.next, k) + 1;
  if (i == k) {
    System.out.println(head.data);
  }
  return i;
}
```

当然，只有得到面试官的首肯，这个解法才算有效。

● 方法B：使用C++

第二种解法是使用C++，并通过引用传值。这样一来，我们就可以返回结点值，而且也能通过传递指针更新计数器。

```
node* nthToLast(node* head, int k, int& i) {
  if (head == NULL) {
    return NULL;
  }
  node * nd = nthToLast(head->next, k, i);
  i = i + 1;
  if (i == k) {
    return head;
  }
  return nd;
}
```

● 方法C：创建包裹类

前面提到，这里的难点在于我们无法同时返回计数器和索引值。如果用一个简单的类（或一个单元素数组）包裹计数器值，就可以模仿按引用传递。

```
public class IntWrapper {
  public int value = 0;
}

LinkedListNode nthToLastR2(LinkedListNode head, int k,
                           IntWrapper i) {
  if (head == null) {
    return null;
  }
  LinkedListNode node = nthToLastR2(head.next, k, i);
  i.value = i.value + 1;
  if (i.value == k) { // 找到倒数第k个元素
    return head;
  }
  return node;
}
```

因为有递归调用，这些递归解法都需要占用$O(n)$空间。

还有不少其他解法这里并未提及。我们可以将计数器存放在静态变量中，或者，可以创建一个类，存放结点和计数器，并返回这个类的实例。不论选用哪种解法，我们都要设法更新结点和计数器，并在每层递归调用的栈都能访问到。

解法 3：迭代法

一种效率更高但不太直观的解法是以迭代方式实现。我们可以使用两个指针`p1`和`p2`，并将它们指向链表中相距*k*个结点的两个结点，具体做法是先将`p1`和`p2`指向链表首结点，然后将`p2`向前移动*k*个结点。之后，我们以相同的速度移动这两个指针，`p2`会在移动`LENGTH - k`步后抵达链表尾结点。此时，`p1`会指向链表第`LENGTH - k`个结点，或者说倒数第*k*个结点。

下面的代码实现了该算法。

```
LinkedListNode nthToLast(LinkedListNode head, int k) {
   if (k <= 0) return null;

   LinkedListNode p1 = head;
   LinkedListNode p2 = head;

   // p2向前移动k个结点
   for (int i = 0; i < k - 1; i++) {
     if (p2 == null) return null; // 错误检查
     p2 = p2.next;
   }
   if (p2 == null) return null;

   /* 现在以同样的速度移动p1和p2，当p2抵达链表末尾时，
    * p1刚好指向倒数第k个结点 */
   while (p2.next != null) {
     p1 = p1.next;
     p2 = p2.next;
   }
   return p1;
}
```

这个算法的时间复杂度为$O(n)$，空间复杂度为$O(1)$。

2.3　实现一个算法，删除单向链表中间的某个结点，假定你只能访问该结点。（第 48 页）

解法

在这个问题中，你访问不到链表的首结点，只能访问那个待删除结点。解法很简单，直接将后继结点的数据复制到当前结点，然后删除这个后继结点。

下面是该算法的实现代码。

```
public static boolean deleteNode(LinkedListNode n) {
   if (n == null || n.next == null) {
     return false; // 失败
   }
   LinkedListNode next = n.next;
   n.data = next.data;
   n.next = next.next;
   return true;
}
```

注意，若待删除结点为链表的尾结点，则这个问题无解。没关系，面试官就是想要你指出这一点，并讨论该怎么处理这种情况。例如，你可以考虑将该结点标记为假的。

2.4 编写代码，以给定值*x*为基准将链表分割成两部分，所有小于*x*的结点排在大于或等于*x*的结点之前。（第 49 页）

解法

要是链表换作数组，搬移元素时就要特别小心，因为搬移数组元素的开销很大。

不过，移动链表的元素则要容易许多。我们不必移动和交换元素，可以直接创建两个链表：一个链表存放小于*x*的元素；另一个链表存放大于或等于*x*的元素。

我们会迭代访问整个链表，将元素插入`before`或`after`链表。一旦抵达链表末端，则表明拆分完成，最后合并两个链表。

下面是该方法的实现代码。

```
/* 传入链表的首结点，以及作为链表分割
 * 基准的值 */
public LinkedListNode partition(LinkedListNode node, int x) {
   LinkedListNode beforeStart = null;
   LinkedListNode beforeEnd = null;
   LinkedListNode afterStart = null;
   LinkedListNode afterEnd = null;

   /* 分割链表 */
   while (node != null) {
     LinkedListNode next = node.next;
     node.next = null;
     if (node.data < x) {
        /* 将结点插入before链表 */
        if (beforeStart == null) {
           beforeStart = node;
           beforeEnd = beforeStart;
        } else {
           beforeEnd.next = node;
           beforeEnd = node;
        }
     } else {
        /* 将结点插入after链表 */
        if (afterStart == null) {
           afterStart = node;
           afterEnd = afterStart;
        } else {
           afterEnd.next = node;
           afterEnd = node;
        }
     }
     node = next;
   }

   if (beforeStart == null) {
     return afterStart;
   }

   /* 合并before和after链表 */
```

```
    beforeEnd.next = afterStart;
    return beforeStart;
}
```

为了追踪两个链表却要维护四个变量，你可能觉得有点碍眼，不少人都有同感。我们可以移除其中部分变量，不过代码执行效率会略打折扣。效率降低的原因在于遍历整个链表的时间略微延长。不过，大O表示的时间复杂度仍保持不变，同时代码也变得更简短、扼要。

第二种解法略有不同。结点不再追加至`before`和`after`链表的末端，而是插入这两个链表的前端。

```
1  public LinkedListNode partition(LinkedListNode node, int x) {
2     LinkedListNode beforeStart = null;
3     LinkedListNode afterStart = null;
4
5     /* 分割链表 */
6     while (node != null) {
7       LinkedListNode next = node.next;
8       if (node.data < x) {
9          /* 将结点插入before链表的前端 */
10         node.next = beforeStart;
11         beforeStart = node;
12      } else {
13         /* 将结点插入after链表的前端 */
14         node.next = afterStart;
15         afterStart = node;
16      }
17      node = next;
18    }
19
20    /* 合并before链表和after链表 */
21    if (beforeStart == null) {
22      return afterStart;
23    }
24
25    /* 定位至before链表末尾，合并两个链表 */
26    LinkedListNode head = beforeStart;
27    while (beforeStart.next != null) {
28      beforeStart = beforeStart.next;
29    }
30    beforeStart.next = afterStart;
31
32    return head;
33 }
```

注意，解决这个问题时，必须非常小心地处理`null`值。再看看上面第7行代码，为什么要有这行代码？这是因为在循环访问链表时，也会修改这个链表。我们必须用临时变量记下后继结点，这样才能知道下一次迭代要用到该后继结点。

2.5 给定两个用链表表示的整数，每个结点包含一个数位。这些数位是反向存放的，也就是个位排在链表首部。编写函数对这两个整数求和，并用链表形式返回结果。

进阶

假设这些数位是正向存放的，请再做一遍。（第 49 页）

解法

着手解决这个问题之前，有必要回顾一下加法是怎么回事，比如：

```
  6 1 7
+ 2 9 5
```

首先，7加5得到12。其中，2为结果12的个位，1则为十位相加时的进位。然后，将1、1和9相加，得到11。十位数字为1，另一个1则成为下一步运算的进位。最后，将1、6和2相加得到9。因此，这两个整数求和的结果为912。

我们可以递归地模拟这个过程，将两个结点的值逐一相加，如有进位则转入下一个结点。下面以两个链表为例进行说明：

```
  7 -> 1 -> 6
+ 5 -> 9 -> 2
```

步骤如下。

(1) 首先，将7和5相加，结果为12，则2成为结果链表的第一个结点，并将1进位给下一次求和运算。

```
链表：2 -> ?
```

(2) 然后，将1、9和上面的进位相加，结果为11，于是1成为结果链表的第二个元素，另一个1则进位给下一个求和运算。

```
链表：2 -> 1 -> ?
```

(3) 最后，将6、2和上面的进位相加，得到9，同时也成为结果链表的最后一个元素。

```
链表：2 -> 1 -> 9
```

下面是该算法的实现代码。

```
LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2,
                        int carry) {
  /* 两个链表全部为空且进位为0，则函数返回 */
  if (l1 == null && l2 == null && carry == 0) {
    return null;
  }

  LinkedListNode result = new LinkedListNode();

  /* 将value以及l1和l2的data相加 */
  int value = carry;
  if (l1 != null) {
    value += l1.data;
```

```
    }
    if (l2 != null) {
      value += l2.data;
    }

    result.data = value % 10; /* 求和结果的个位 */

    /* 递归 */
    LinkedListNode more = addLists(l1 == null ? null : l1.next,
                                   l2 == null ? null : l2.next,
                                   value >= 10 ? 1 : 0);
    result.setNext(more);
    return result;
}
```

在实现这段代码时，务必注意处理一个链表比另一个链表结点少的情况。我们可不想碰到空指针异常。

进阶

从概念上来说，第二部分并无不同（递归，进位处理），但在实现时稍微复杂一些。

(1) 一个链表的结点可能比另一个链表的少，我们无法直接处理这种情况。例如，假设要对`(1 -> 2 -> 3 -> 4)`与`(5 -> 6 -> 7)`求和。务必注意，5应该与2而不是1配对。对此，我们可以一开始先比较两个链表的长度并用零填充较短的链表。

(2) 在前一个问题中，相加的结果不断追加到链表尾部（也即向前传递）。这就意味着递归调用会传入进位，而且会返回结果（随后追加至链表尾部）。不过，这里的结果要加到首部（也即向后传递）。跟前一个问题一样，递归调用必须返回结果和进位。实现也不是太难，但处理起来会更难一些，可以通过创建一个`PartialSum`包裹类来解决这一点。

下面是该算法的实现代码。

```
public class PartialSum {
   public LinkedListNode sum = null;
   public int carry = 0;
}

LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
   int len1 = length(l1);
   int len2 = length(l2);

   /* 用零填充较短的链表，参看注解（1） */
   if (len1 < len2) {
     l1 = padList(l1, len2 - len1);
   } else {
     l2 = padList(l2, len1 - len2);
   }

   /* 对两个链表求和 */
   PartialSum sum = addListsHelper(l1, l2);

   /* 如有进位，则插入链表首部，否则，直接返回
```

```
    * 整个链表 */
   if (sum.carry == 0) {
     return sum.sum;
   } else {
     LinkedListNode result = insertBefore(sum.sum, sum.carry);
     return result;
   }
}

PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
   if (l1 == null && l2 == null) {
     PartialSum sum = new PartialSum();
     return sum;
   }
   /* 对较小数字递归求和 */
   PartialSum sum = addListsHelper(l1.next, l2.next);

   /* 将进位和当前数据相加 */
   int val = sum.carry + l1.data + l2.data;

   /* 插入当前数字的求和结果 */
   LinkedListNode full_result = insertBefore(sum.sum, val % 10);

   /* 返回求和结果和进位值 */
   sum.sum = full_result;
   sum.carry = val / 10;
   return sum;
}

/* 用零填充链表 */
LinkedListNode padList(LinkedListNode l, int padding) {
   LinkedListNode head = l;
   for (int i = 0; i < padding; i++) {
     LinkedListNode n = new LinkedListNode(0, null, null);
     head.prev = n;
     n.next = head;
     head = n;
   }
   return head;
}

/* 辅助函数，将结点插入链表首部 */
LinkedListNode insertBefore(LinkedListNode list, int data) {
   LinkedListNode node = new LinkedListNode(data, null, null);
   if (list != null) {
     list.prev = node;
     node.next = list;
   }
   return node;
}
```

注意，上面的代码已将`insertBefore()`、`padList()`和`length()`（未列出）单列为独立方法。这样一来，代码更清晰更易读，在面试时这么做非常可取！

2.6　给定一个有环链表，实现一个算法返回环路的开头结点。（第 49 页）

解法

这个问题是由经典面试题——检测链表是否存在环路——演变而来。下面我们将运用模式匹配法来解决这个问题。

第 1 部分：检测链表是否存在环路

检测链表是否存在环路，有一种简单的做法叫FastRunner/SlowRunner法。FastRunner一次移动两步，而SlowRunner一次移动一步。这就好比两辆赛车绕着同一条赛道以不同的速度前进，最终必然会碰到一起。

聪明的读者可能会问：FastRunner会不会刚好“越过”SlowRunner，而不发生碰撞呢？绝无可能。假设FastRunner真的越过了SlowRunner，且SlowRunner处于位置i，FastRunner处于位置i + 1。那么，在前一步，SlowRunner就处于位置i - 1，FastRunner处于位置((i + 1) - 2)或i - 1。也就是说，两者碰在一起了。

第 2 部分：什么时候碰在一起？

假定这个链表有一部分不存在环路，长度为k。

若运用第1部分的算法，FastRunner和SlowRunner什么时候会碰在一起呢？

我们知道，SlowRunner每走p步，FastRunner就会走2p步。因此，当SlowRunner走了k步进入环路部分时，FastRunner已走了总共2k步，进入环路部分已有2k - k步或k步。由于k可能比环路长度大得多，实际上我们应该将它写作mod(k, LOOP_SIZE)步，并用K表示。

对于之后的每一步，FastRunner和SlowRunner之间不是走远一步就是更近一步，具体要看观察的角度。也就是说，因为两者处于圆圈中，当A以远离B的方向走出q步时，同时也是向B更近了q步。综上，我们得出以下几点：

(1) SlowRunner处于环路中的0步位置；

(2) FastRunner处于环路中的K步位置；

(3) SlowRunner落后于FastRunner，相距K步；

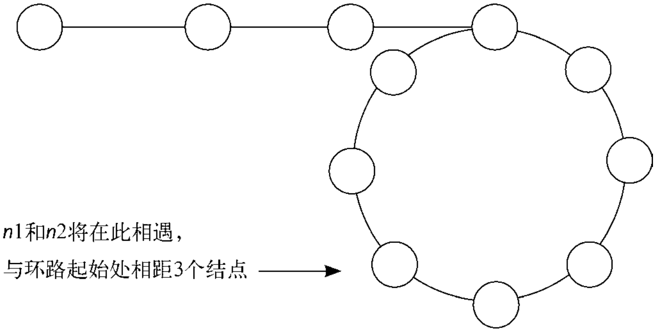
(4) FastRunner落后于SlowRunner，相距LOOP_SIZE - K步；

(5) 每过一个单位时间，FastRunner就会更接近SlowRunner一步。

那么，两个结点什么时候相遇？若FastRunner落后于SlowRunner，相距LOOP_SIZE - K步，并且每经过一个单位时间，FastRunner就走近SlowRunner一步，那么，两者将在LOOP_SIZE - K步之后相遇。此时，两者与环路起始处相距K步，我们将这个位置称为CollisionSpot。

第 3 部分：如何找到环路起始处？

现在我们知道CollisionSpot与环路起始处相距K个结点。由于K = mod(k, LOOP_SIZE)（或者换句话说，k = K + M * LOOP_SIZE，其中M为任意整数），也可以说，CollisionSpot与环路起始处相距k个结点。例如，若有个环路长度为5个结点，有个结点N处于距离环路起始处2个结点的地方，我们也可以换个说法：这个结点处于距离环路起始处7个、12个甚至397个结点。



至此，CollisionSpot和LinkedListHead与环路起始处均相距k个结点。

现在，若用一个指针指向CollisionSpot，用另一个指针指向LinkedListHead，两者与LoopStart均相距k个结点。以同样的速度移动，这两个指针会再次碰在一起——这次是在k步之后，此时两个指针都指向LoopStart，然后只需返回该结点即可。

第 4 部分：将全部整合在一起

总结一下，FastPointer的移动速度是SlowPointer的两倍。当SlowPointer走了k个结点进入环路时，FastPointer已进入链表环路k个结点。也就是说FastPointer和SlowPointer相距LOOP_SIZE - k个结点。

接下来，若SlowPointer每走一个结点，FastPointer就走两个结点，每走一次，两者的距离就会更近一个结点。因此，在走了LOOP_SIZE - k次后，它们就会碰在一起。这时两者距离环路起始处有k个结点。

链表首部与环路起始处也相距k个结点。因此，若其中一个指针保持不变，另一个指针指向链表首部，则两个指针就会在环路起始处相会。

根据第1、2、3部分，就能直接导出下面的算法。

(1) 创建两个指针：FastPointer和SlowPointer。

(2) SlowPointer每走一步，FastPointer就走两步。

(3) 两者碰在一起时，将SlowPointer指向LinkedListHead，FastPointer保持不变。

(4) 以相同速度移动SlowPointer和FastPointer，一次一步，然后返回新的碰撞处。

下面是该算法的实现代码。

```
LinkedListNode FindBeginning(LinkedListNode head) {
   LinkedListNode slow = head;
   LinkedListNode fast = head;

   /* 找出碰撞处，将处于链表中LOOP_SIZE - k步的
    * 位置 */
   while (fast != null && fast.next != null) {
     slow = slow.next;
     fast = fast.next.next;
     if (slow == fast) { // 碰撞
        break;
     }
```

```
  }

  /* 错误检查，没有碰撞处，也即没有环路 */
  if (fast == null || fast.next == null) {
    return null;
  }

  /* 将slow指向首部，fast指向碰撞处，两者
   * 距离环路起始处k步，若两者以相同速度移动，
   * 则必定会在环路起始处碰在一起 */
  slow = head;
  while (slow != fast) {
    slow = slow.next;
    fast = fast.next;
  }

  /* 至此两者均指向环路起始处 */
  return fast;
}
```

2.7　编写一个函数，检查链表是否为回文。（第 49 页）

解法

要解决这个问题，可以将回文（palindrome）定义为`0 -> 1 -> 2 -> 1 -> 0`。显然，若链表是回文，不管正着看还是反着看，都是一样的。由此可以得出第一种解法。

解法 1：反转并比较

第一种解法是反转整个链表，然后比较反转链表和原始链表。若两者相同，则该链表为回文。

注意，在比较原始链表和反转链表时，其实只需比较链表的前半部分。若原始链表和反转链表的前半部分相同，那么，两者的后半部分肯定相同。

解法 2：迭代法

要想探测链表的前半部分是否为后半部分反转而成，该怎么做呢？只需将链表前半部分反转，可以利用栈来实现。

我们需要将前半部分结点入栈。根据链表长度已知与否，入栈有两种方式。

若链表长度已知，可以用标准for迭代访问前半部分结点，将每个结点入栈。当然，要小心处理链表长度为奇数的情况。

若链表长度未知，可以利用本章开头描述的快慢runner方法迭代访问链表。在迭代循环的每一步，将慢速runner的数据入栈。在快速runner抵达链表尾部时，慢速runner刚好位于链表中间位置。至此，栈里就存放了链表前半部分的所有结点，不过顺序是相反的。

接下来，我们只需迭代访问链表余下结点。每次迭代时，比较当前结点和栈顶元素，若完成迭代时比较结果完全相同，则该链表是回文序列。

```
boolean isPalindrome(LinkedListNode head) {
  LinkedListNode fast = head;
```

```
  LinkedListNode slow = head;

  Stack<Integer> stack = new Stack<Integer>();

  /* 将链表前半部分元素入栈。当快速runner（移动速度为
   * 慢速runner的两倍）到达链表尾部时，则慢速runner已
   * 处在链表中间位置 */
  while (fast != null && fast.next != null) {
    stack.push(slow.data);
    slow = slow.next;
    fast = fast.next.next;
  }

  /* 链表有奇数个元素，跳过中间元素 */
  if (fast != null) {
    slow = slow.next;
  }

  while (slow != null) {
    int top = stack.pop().intValue();

    /* 两者不相同，则该链表不是回文序列 */
    if (top != slow.data) {
       return false;
    }
    slow = slow.next;
  }
  return true;
}
```

解法 3：递归法

首先，简要介绍下面的解法用到的记号：用记号Kx表示结点时，变量K指示结点数据的值，而x（取f或b）指示该结点是值为K的前方结点还是后方结点。例如，在下面的链表中，结点3b指的是值为3的第二个（b → back，即后方）结点。

接下来，跟许多链表问题一样，可以用递归法解决这个问题。我们靠直觉可能就会想到要比较元素0和元素n，元素1和元素n-1，元素2和元素n-2，等等，直至中间元素。

例如：

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

为了运用这种方法，首先必须知道什么时候到达中间元素，这也形成了递归的终止条件。每次递归调用传入`length - 2`为长度，当长度等于0或1时，表明当前已处于链表中间位置。

```
recurse(Node n, int length) {
  if (length == 0 || length == 1) {
    return [something]; // 中间
  }
  recurse(n.next, length - 2);
  ...
}
```

这个方法构成了`isPalindrome`方法的轮廓，而该算法的实质则是比较结点`i`和结点`n - i`，检查链表是否为回文序列。具体该怎么做呢？

仔细分析下面的调用栈：

```
1 v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2    v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3      v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
4        v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5        returns v3
6      returns v2
7    returns v1
8 returns ?
```

在上面的调用栈中，每次调用都会比较其首结点和链表后半部分对应结点，检查链表是否为回文序列。即：

- 第1行需要比较结点`0f`和结点`0b`；
- 第2行需要比较结点`1f`和结点`1b`；
- 第3行需要比较结点`2f`和结点`2b`；
- 第4行需要比较结点`3f`和结点`3b`。

若将上面的栈倒过来，按如下顺序传回结点，我们只需：

- 第4行发现传入结点为中间结点（因为`length = 1`），传回`head.next`。其中`head`为结点3，因此`head.next`为结点`2b`；
- 第3行比较首部即结点`2f`和`returned_node`（上次递归调用返回的值）即结点`2b`。若两个结点的值相等，则传回结点`1b`的引用（`returned_node.next`）；
- 第2行比较首部（结点`1f`）和`returned_node`（结点`1b`）。若两个结点的值相等，则传回结点`0b`的引用（或`returned_node.next`）；
- 第1行比较首部（结点`0f`）和`returned_node`（结点`0b`）。若两个结点的值相等，则返回ture。

归纳一下，每次调用都会比较其首部和`returned_node`，然后回传`returned_node.next`。最终每个结点`i`都会与结点`n - i`进行比较。只要有任意一对结点的值不相等，就立即返回`false`，调用栈的上一级调用都会检查这个布尔值。

但是等等，你可能会问，我们一会儿说要返回一个布尔值，一会儿说要返回一个结点？到底返回什么？

两个都要返回。我们创建了一个包含布尔值和结点两个成员的简单类，调用时只需返回该类的实例。

```
class Result {
   public LinkedListNode node;
   public boolean result;
}
```

下面举例说明示例链表每次递归调用的参数和返回值。

```
isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
   isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
```

```
    isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
      isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
        isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
        returns node 3b, true
      returns node 2b, true
    returns node 1b, true
  returns node 0b, true
returns nobe 0b, true
```

至此，这段代码实现起来很简单，只需填入细节即可。

```
Result isPalindromeRecurse(LinkedListNode head, int length) {
   if (head == null || length == 0) {
     return new Result(null, true);
   } else if (length == 1) {
     return new Result(head.next, true);
   } else if (length == 2) {
     return new Result(head.next.next,
                       head.data == head.next.data);
   }
   Result res = isPalindromeRecurse(head.next, length - 2);
   if (!res.result || res.node == null) {
     return res;
   } else {
     res.result = head.data == res.node.data;
     res.node = res.node.next;
     return res;
   }
}

boolean isPalindrome(LinkedListNode head) {
   Result p = isPalindromeRecurse(head, listSize(head));
   return p.result;
}
```

有些人可能会有疑问，为什么要这么费力专门创建一个Result类，有没有更好的办法？还真没有，至少用Java实现的话没有。

然而，若用C或C++实现的话，我们可以传入一个指针的指针。

```
bool isPalindromeRecurse(Node head, int length, Node** next) {
   ...
}
```

代码不太好看，但行之有效。

9.3 栈与队列

3.1 描述如何只用一个数组来实现三个栈。（第 50 页）

解法

和许多问题一样，这个问题的解法基本上取决于你要对栈支持到什么程度。若每个栈分配固

定大小的空间，就能满足需要，那么照做便是。不过，这么做的话，有可能其中一个栈的空间不够用了，而同时其他的栈却几乎是空的。 另一种做法是弹性处理栈的空间分配，但这么一来，这个问题的复杂度又会大大增加。

方法 1：固定分割

我们可以将整个数组划分为三等份，将每个栈的增长限制在各自的空间里。注意：记号“[”表示包含端点，“(”表示不包含端点。

- 栈1，使用[0, n/3)。
- 栈2，使用[n/3, 2n/3)。
- 栈3，使用[2n/3, n)。

下面是该解法的实现代码。

```
int stackSize = 100;
int[] buffer = new int [stackSize * 3];
int[] stackPointer = {-1, -1, -1}; // 用于追踪栈顶元素的指针

void push(int stackNum, int value) throws Exception {
  /* 检查有无空闲空间 */
  if (stackPointer[stackNum] + 1 >= stackSize) { // 最后一个元素
    throw new Exception(“Out of space.”);
  }
  /* 栈指针自增，然后更新栈顶元素的值 */
  stackPointer[stackNum]++;
  buffer[absTopOfStack(stackNum)] = value;
}

int pop(int stackNum) throws Exception {
  if (stackPointer[stackNum] == -1) {
    throw new Exception(“Trying to pop an empty stack.”);
  }
  int value = buffer[absTopOfStack(stackNum)]; // 获取栈顶元素的值
  buffer[absTopOfStack(stackNum)] = 0; // 清零指定索引元素的值
  stackPointer[stackNum]--; // 指针自减
  return value;
}

int peek(int stackNum) {
  int index = absTopOfStack(stackNum);
  return buffer[index];
}

boolean isEmpty(int stackNum) {
  return stackPointer[stackNum] == -1;
}

/* 返回栈“stackNum”栈顶元素的索引，绝对量 */
int absTopOfStack(int stackNum) {
  return stackNum * stackSize + stackPointer[stackNum];
}
```

如果知道与这些栈的使用情况相关的更多信息，就可以对上面的算法做相应的改进。例如，若预估Stack 1的元素比Stack 2多很多，那么，就可以给Stack 1多分配一点空间，给Stack 2少分配一些空间。

方法 2：弹性分割

第二种做法是允许栈块的大小灵活可变。当一个栈的元素个数超出其初始容量时，就将这个栈扩容至许可的容量，必要时还要搬移元素。

此外，我们会将数组设计成环状的，最后一个栈可能从数组末尾开始，环绕到数组开头。

请注意，这种解法的代码远比面试中常见的要复杂得多。你可以试着提供伪码，或是其中某几部分的代码，但要完整实现的话，难度就有点大了。

```
/* StackData是个简单的类，存放每个栈相关的数据，
 * 但并未存放栈的实际元素*/
public class StackData {
   public int start;
   public int pointer;
   public int size = 0;
   public int capacity;
   public StackData(int _start, int _capacity) {
     start = _start;
     pointer = _start - 1;
     capacity = _capacity;
   }

   public boolean isWithinStack(int index, int total_size) {
     /* 注意：如果栈回绕了，首部（右侧）回绕到
      * 左边 */
     if (start <= index && index < start + capacity) {
        // 不回绕，或回绕时的“首部”（右侧）
        return true;
     } else if (start + capacity > total_size &&
                 index < (start + capacity) % total_size) {
        // 回绕时的尾部（左侧）
        return true;
     }
     return false;
   }
}

public class QuestionB {
   static int number_of_stacks = 3;
   static int default_size = 4;
   static int total_size = default_size * number_of_stacks;
   static StackData [] stacks = {new StackData(0, default_size),
     new StackData(default_size, default_size),
     new StackData(default_size * 2, default_size)};
   static int [] buffer = new int [total_size];

   public static void main(String [] args) throws Exception  {
     push(0, 10);
```

```
    push(1, 20);
    push(2, 30);
    int v = pop(0);
    ...
  }

  public static int numberOfElements() {
    return stacks[0].size + stacks[1].size + stacks[2].size;
  }

  public static int nextElement(int index) {
    if (index + 1 == total_size) return 0;
    else return index + 1;
  }

  public static int previousElement(int index) {
    if (index == 0) return total_size - 1;
    else return index - 1;
  }

  public static void shift(int stackNum) {
    StackData stack = stacks[stackNum];
    if (stack.size >= stack.capacity) {
       int nextStack = (stackNum + 1) % number_of_stacks;
       shift(nextStack); // 腾出若干空间
       stack.capacity++;
    }

    // 以相反顺序搬移元素
    for (int i = (stack.start + stack.capacity - 1) % total_size;
        stack.isWithinStack(i, total_size);
        i = previousElement(i)) {
       buffer[i] = buffer[previousElement(i)];
    }

    buffer[stack.start] = 0;
    stack.start = nextElement(stack.start); // 移动栈的起始位置
    stack.pointer = nextElement(stack.pointer); // 移动指针
    stack.capacity--; // 恢复到原先的容量
  }

  /* 搬移到其他栈上，以扩大栈的容量 */
  public static void expand(int stackNum) {
    shift((stackNum + 1) % number_of_stacks);
    stacks[stackNum].capacity++;
  }

  public static void push(int stackNum, int value)
       throws Exception {
    StackData stack = stacks[stackNum];
    /* 检查空间够不够 */
    if (stack.size >= stack.capacity) {
       if (numberOfElements() >= total_size) { // 全部都满了
         throw new Exception("Out of space.");
```

```
      } else { // 只是需要搬移元素
        expand(stackNum);
      }
    }
    /* 找出顶端元素在数组中的索引值，并加1，
     * 并增加栈指针 */
    stack.size++;
    stack.pointer = nextElement(stack.pointer);
    buffer[stack.pointer] = value;
  }

  public static int pop(int stackNum) throws Exception {
    StackData stack = stacks[stackNum];
    if (stack.size == 0) {
       throw new Exception("Trying to pop an empty stack.");
    }
    int value = buffer[stack.pointer];
    buffer[stack.pointer] = 0;
    stack.pointer = previousElement(stack.pointer);
    stack.size--;
    return value;
  }

  public static int peek(int stackNum) {
    StackData stack = stacks[stackNum];
    return buffer[stack.pointer];
  }

  public static boolean isEmpty(int stackNum) {
    StackData stack = stacks[stackNum];
    return stack.size == 0;
  }
}
```

遇到类似的问题，应该力求编写清晰、可维护的代码，这很重要。你应该引入额外的类，比如这里使用了`StackData`，并将大块代码独立为单独的方法。当然，这个建议同样适用于真正的软件开发。

3.2 请设计一个栈，除pop与push方法，还支持min方法，可返回栈元素中的最小值。push、pop和min三个方法的时间复杂度必须为$O(1)$。（第 50 页）

解法

既然是最小值，就不会经常变动，只有在更小的元素加入时，才会改变。

一种解法是在`Stack`类里添加一个`int`型的`minValue`。当`minValue`出栈时，我们会搜索整个栈，找出新的最小值。可惜，这不符合入栈和出栈操作时间为$O(1)$的要求。

为进一步理解这个问题，下面用一个简短的例子加以说明：

```
push(5); // 栈为{5}，最小值为5
push(6); // 栈为{6, 5}，最小值为5
push(3); // 栈为{3, 6, 5}，最小值为3
```

```
push(7); // 栈为{7, 3, 6, 5}，最小值为3
pop(); // 弹出7，栈为{3, 6, 5}，最小值为3
pop(); // 弹出3，栈为{6, 5}，最小值为5
```

注意观察，当栈回到之前的状态（{6, 5}）时，最小值也回到之前的状态（5），这就导出了我们的第二种解法。

只要记下每种状态的最小值，我们就能轻易获取最小值。实现很简单，每个结点记录当前最小值即可。这么一来，要找到min，直接查看栈顶元素就能得到最小值。

当一个元素入栈时，该元素会记下当前最小值，将min记录在自身数据结构的min成员中。

```
public class StackWithMin extends Stack<NodeWithMin> {
   public void push(int value) {
     int newMin = Math.min(value, min());
     super.push(new NodeWithMin(value, newMin));
   }

   public int min() {
     if (this.isEmpty()) {
        return Integer.MAX_VALUE; // 错误值
     } else {
        return peek().min;
     }
   }
}

class NodeWithMin {
   public int value;
   public int min;
   public NodeWithMin(int v, int min){
     value = v;
     this.min = min;
   }
}
```

但是，这种做法有个缺点：当栈很大时，每个元素都要记录min，就会浪费大量空间。还有没有更好的做法？

利用额外的栈来记录这些min，我们也许可以比之前做得更好一点。

```
public class StackWithMin2 extends Stack<Integer> {
   Stack<Integer> s2;
   public StackWithMin2() {
     s2 = new Stack<Integer>();
   }

   public void push(int value){
     if (value <= min()) {
       s2.push(value);
     }
     super.push(value);
   }

   public Integer pop() {
```

```
    int value = super.pop();
    if (value == min()) {
       s2.pop();
    }
    return value;
  }

  public int min() {
    if (s2.isEmpty()) {
       return Integer.MAX_VALUE;
    } else {
       return s2.peek();
    }
  }
}
```

为什么这么做可以节省空间？假设有个很大的栈，而第一个元素刚好是最小值。对于第一种解法，我们需要记录n个整数，其中n为栈的大小。不过，对于第二种解法，我们只需存储几项数据：第二个栈（只有一个元素），以及栈本身数据结构的若干成员。

3.3　设想有一堆盘子，堆太高可能会倒下来。因此，在现实生活中，盘子堆到一定高度时，我们就会另外堆一堆盘子。请实现数据结构SetOfStacks，模拟这种行为。SetOfStacks应该由多个栈组成，并且在前一个栈填满时新建一个栈。此外，SetOfStacks.push()和SetOfStacks.pop()应该与普通栈的操作方法相同（也就是说，pop()返回的值，应该跟只有一个栈时的情况一样）。

进阶

实现一个popAt(int index)方法，根据指定的子栈，执行pop操作。（第 51 页）

解法

在这个问题中，根据题意，数据结构应该类似如下：

```
class SetOfStacks {
   ArrayList<Stack> stacks = new ArrayList<Stack>();
   public void push(int v) { ... }
   public int pop() { ... }
}
```

其中push()的行为必须跟单一栈的一样，这就意味着push()要对栈数组的最后一个栈调用push()。不过，这里处理起来必须小心一点：若最后一个栈被填满，就需新建一个栈。实现代码大致如下：

```
public void push(int v) {
   Stack last = getLastStack();
   if (last != null && !last.isFull()) { // 添加到最后一个栈中
     last.push(v);
   } else { // 必须新建一个栈
     Stack stack = new Stack(capacity);
     stack.push(v);
     stacks.add(stack);
```

```
    }
}
```

那么，pop()该怎么做？它的行为与push()类似，也就是说，应该操作最后一个栈。若最后一个栈为空（执行出栈操作后），就必须从栈数组中移除这个栈。

```
public int pop() {
   Stack last = getLastStack();
   int v = last.pop();
   if (last.size == 0) stacks.remove(stacks.size() - 1);
   return v;
}
```

进阶：实现popAt(int index)

这个实现起来有点棘手，不过，我们可以设想一个“推入”动作。从栈1弹出元素时，我们需要移出栈2的栈底元素，并将其推到栈1中。随后，将栈3的栈底元素推入栈2，将栈4的栈底元素推入栈3，等等。

你可能会指出，何必执行“推入”操作，有些栈不填满不是也挺好的。而且，这还会改善时间复杂度（元素很多时尤其明显），但是，若之后有人假定所有的栈（最后一个栈除外）都是填满的，就可能会让我们陷入棘手的境地。这个问题并没有“标准答案”，你应该跟面试官讨论各种做法的优劣。

```
public class SetOfStacks {
   ArrayList<Stack> stacks = new ArrayList<Stack>();
   public int capacity;
   public SetOfStacks(int capacity) {
     this.capacity = capacity;
   }

   public Stack getLastStack() {
     if (stacks.size() == 0) return null;
     return stacks.get(stacks.size() - 1);
   }

   public void push(int v) { /* 参看之前的代码 */ }
   public int pop() { /* 参看之前的代码 */ }
   public boolean isEmpty() {
     Stack last = getLastStack();
     return last == null || last.isEmpty();
   }

   public int popAt(int index) {
     return leftShift(index, true);
   }

   public int leftShift(int index, boolean removeTop) {
     Stack stack = stacks.get(index);
     int removed_item;
     if (removeTop) removed_item = stack.pop();
     else removed_item = stack.removeBottom();
```

```
     if (stack.isEmpty()) {
        stacks.remove(index);
     } else if (stacks.size() > index + 1) {
        int v = leftShift(index + 1, false);
        stack.push(v);
     }
     return removed_item;
   }
}

public class Stack {
   private int capacity;
   public Node top, bottom;
   public int size = 0;

   public Stack(int capacity) { this.capacity = capacity; }
   public boolean isFull() { return capacity == size; }

   public void join(Node above, Node below) {
     if (below != null) below.above = above;
     if (above != null) above.below = below;
   }

   public boolean push(int v) {
     if (size >= capacity) return false;
     size++;
     Node n = new Node(v);
     if (size == 1) bottom = n;
     join(n, top);
     top = n;
     return true;
   }

   public int pop() {
     Node t = top;
     top = top.below;
     size--;
     return t.value;
   }

   public boolean isEmpty() {
     return size == 0;
   }

   public int removeBottom() {
     Node b = bottom;
     bottom = bottom.above;
     if (bottom != null) bottom.below = null;
     size--;
     return b.value;
   }
}
```

这个问题在概念上并不是很难，但要完整实现需要编写大量代码。面试官一般不会要求你写

出全部代码。

解决这类问题，有个很好的策略，就是尽量将代码分离出来，写成独立的方法，比如`popAt`可以调用的`leftShift`。这样一来，你的代码就会更加清晰，而你在处理细节之前，也有机会先铺设好代码的骨架。

3.4 在经典问题汉诺塔中，有 3 根柱子及*N*个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自底向上从大到小依次套在第一根柱子上（即每一个盘子只能放在更大的盘子上面）。移动圆盘时有以下限制：

(1) 每次只能移动一个盘子；

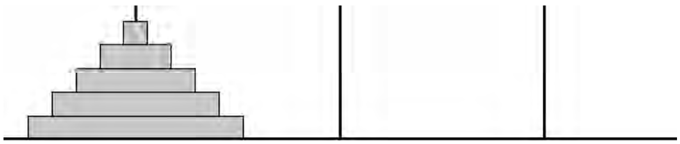
(2) 盘子只能从柱子顶端滑出移到下一根柱子；

(3) 盘子只能叠在比它大的盘子上。

请运用栈，编写程序将所有盘子从第一根柱子移到最后一根柱子。（第 51 页）

解法

这个问题看起来很适合采用基本案例构建法。



我们先从最简单的例子$n = 1$开始。

当$n = 1$时，能否将盘子1从柱1移至柱3？回答是肯定的。

直接将盘子1从柱1移至柱3。

当$n = 2$时，能否将盘子1和盘子2从柱1移至柱3？可以。

(1) 将盘子1从柱1移至柱2。

(2) 将盘子2从柱1移至柱3。

(3) 将盘子1从柱2移至柱3。

注意，上述步骤将柱2用作缓冲区，在我们将其他盘子移至柱3时，柱2会暂存一个盘子。

当$n = 3$时，能否将盘子1、2、3从柱1移至柱3？可以。

(1) 从上面可知，我们可以将上面的两个盘子从一根柱子移至另一根柱子，因此假定已经这么做了，只不过，这里是将这两个盘子移至柱2。

(2) 将盘子3移至柱3。

(3) 将盘子1、2移至柱3。重复步骤1即可。

当$n = 4$时，能否将盘子1、2、3、4从柱1移至柱3？可以。

(1) 将盘子1、2、3移至柱2。具体做法参见前面的例子。

(2) 将盘子4移至柱3。

(3) 将盘子1、2、3移至柱3。

注意，柱2和柱3之间并无多大区别，只是叫法不一样，实则是等价的。把柱2作为缓冲，以将盘子移至柱3，相比把柱3用作缓冲，以将盘子移至柱2，并无区别。

根据上述做法，很自然地就可以导出递归算法。在每一部分，我们都会执行以下步骤，用伪码简述如下：

```
moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
   /* 终止条件 */
   if (n <= 0) return;

   /* 将顶端n - 1个盘子从origin移至buffer，
    * 将destination用作缓冲区。 */
   moveDisks(n - 1, origin, buffer, destination);

   /* 将origin顶端的盘子移至destination */
   moveTop(origin, destination);

   /* 将顶部n - 1个盘子从buffer移至destination，
    * 将origin用作缓冲区。 */
   moveDisks(n - 1, buffer, destination, origin);
}
```

下面的代码给出了这个算法更详细的实现，其中还运用了面向对象设计思想。

```
public static void main(String[] args)
   int n = 3;
   Tower[] towers = new Tower[n];
   for (int i = 0; i < 3; i++) {
     towers[i] = new Tower(i);
   }

   for (int i = n - 1; i >= 0; i--) {
     towers[0].add(i);
   }
   towers[0].moveDisks(n, towers[2], towers[1]);
}

public class Tower {
   private Stack<Integer> disks;
   private int index;
   public Tower(int i) {
     disks = new Stack<Integer>();
     index = i;
   }

   public int index() {
     return index;
   }

   public void add(int d) {
     if (!disks.isEmpty() && disks.peek() <= d) {
        System.out.println("Error placing disk " + d);
     } else {
        disks.push(d);
```

9

```
    }
  }

  public void moveTopTo(Tower t) {
    int top = disks.pop();
    t.add(top);
    System.out.println("Move disk " + top + " from " + index() +
                       " to "  + t.index());
  }

  public void moveDisks(int n, Tower destination, Tower buffer) {
    if (n > 0) {
       moveDisks(n - 1, buffer, destination);
       moveTopTo(destination);
       buffer.moveDisks(n - 1, destination, this);
    }
  }
}
```

严格来说，并不一定要将柱子实现为独立的对象，不过，在某种程度上，这么做可使代码更清晰易读。

3.5　实现一个MyQueue类，该类用两个栈来实现一个队列。（第 51 页）

解法

队列和栈的主要区别在于元素进出顺序（先进先出和后进先出），因此，我们需要修改peek()和pop()，以相反顺序执行操作。我们可以利用第二个栈反转元素的次序（弹出s1的元素，压入s2）。在这种实现中，每当执行peek()和pop()操作时，就要将s1的所有元素弹出，压入s2中，然后执行peek/pop操作，再将所有元素压入s1。

上述做法也是可行的，但若连续执行两次pop/peek操作，那么，所有元素都要移来移去，重复移动，这毫无必要。我们可以延迟元素的移动，即让元素一直留在s2中，只有必须反转元素次序时才移动元素。

在这种做法中，stackNewest顶端为最新元素，stackOldest顶端为最旧元素。在将一个元素出列时，我们希望先移除最旧元素，因此先将元素从stackOldest将元素出列。若stackOldest为空，则将stackNewest中的所有元素以相反的顺序转移到stackOldest中。如要插入元素，就将其压入stackNewest，因为最新元素位于它的顶端。

下面是该算法的实现代码。

```
public class MyQueue<T> {
  Stack<T> stackNewest, stackOldest;

  public MyQueue() {
    stackNewest = new Stack<T>();
    stackOldest = new Stack<T>();
  }

  public int size() {
```

```
    return stackNewest.size() + stackOldest.size();
  }

  public void add(T value) {
    /* 压入stackNewest，最新元素始终位于它
     * 的顶端*/
    stackNewest.push(value);
  }

  /* 将元素从stackNewest移至stackOldest，这么做通常是
   * 为了要在stackOldest上执行操作 */
  private void shiftStacks() {
    if (stackOldest.isEmpty()) {
       while (!stackNewest.isEmpty()) {
         stackOldest.push(stackNewest.pop());
       }
    }
  }

  public T peek() {
    shiftStacks(); // 确保stackOldest含有当前元素
    return stackOldest.peek(); // 取回最旧元素
  }

  public T remove() {
    shiftStacks(); // 确保stackOldest含有当前元素
    return stackOldest.pop(); // 弹出最旧元素
  }
}
```

在真正的面试中，你有可能记不清具体的API调用。真的碰到这种情况时，也不必太紧张。你可以问一些小细节，多数面试官都不会为难你。他们更关注你能否做到通盘的理解问题。

3.6　编写程序，按升序对栈进行排序（即最大元素位于栈顶）。最多只能使用一个额外的栈存放临时数据，但不得将元素复制到别的数据结构中（如数组）。该栈支持如下操作：push、pop、peek和isEmpty。**（第 51 页）**

解法

一种做法是实现初步的排序算法。搜索整个栈，找出最小元素，之后将其压入另一个栈。然后，在剩余元素中找出最小的，并将其入栈。这种做法实际上需要三个栈：s1为原先的栈，s2为最终排好序的栈，s3在搜索s1时用作缓冲区。要在s1中搜索最小值，我们需要弹出s1的元素，将它们压入缓冲区s3。

可惜，我们只能使用一个额外的栈。有没有更好的做法？有。

我们不需要反复搜索最小值，若要对s1排序，可以从s1逐一弹出元素，然后按顺序插入s2中。具体怎么做呢？

假设有如下两个栈，其中s2是“排序的”，s1则是未排序的：

s1	s2
	12
5	8
10	3
7	1

从s1中弹出5时，我们需要在s2中找个合适的位置插入这个数。在这个例子中，正确位置是在s2元素3之上。怎样才能将5插入那个位置呢？我们可以先从s1中弹出5，将其存放在临时变量中。然后，将12和8移至s1（从s2中弹出这两个数，并将它们压入s1中），然后将5压入s2。

步骤1

s1	s2
	12
	8
10	3
7	1

tmp = 5

->

步骤2

s1	s2
8	
12	
10	3
7	1

tmp = 5

->

步骤3

s1	s2
8	
12	5
10	3
7	1

tmp = --

注意，8和12仍在s1中，这没关系！对于这两个数，我们可以像处理5那样重复相关步骤，每次弹出s1栈顶元素，将其放入s2中的合适位置。（当然，我们可以将8和12直接从s2移至s1，因为这两个数都比5大，这些元素的“正确位置”就是放在5之上。我们不需要打乱s2的其他元素，当tmp为8或12时，下面代码中的第二个while循环不会执行。）

```
public static Stack<Integer> sort(Stack<Integer> s) {
   Stack<Integer> r = new Stack<Integer>();
   while (!s.isEmpty()) {
     int tmp = s.pop(); //步骤1
     while (!r.isEmpty() && r.peek() > tmp) { //步骤2
       s.push(r.pop());
     }
     r.push(tmp); //步骤3
   }
   return r;
}
```

这个算法的时间复杂度为$O(N^2)$，空间复杂度为$O(N)$。

如果允许使用的栈数量不限，我们可以实现修改版的quicksort或mergesort。

对于mergesort解法，我们可以再创建两个栈，并将这个栈分为两部分。我们会递归排序每个栈，然后将它们归并到一起并排好序，放回原来的栈中。注意，该解法要求每层递归都创建两个额外的栈。

对于quicksort解法，我们会创建两个额外的栈，并根据基准元素（pivot element）将这个栈分为两个栈。这两个栈会进行递归排序，然后归并在一起，放回原来的栈中。与上一个解法一样，每层递归都会创建两个额外的栈。

3.7 有家动物收容所只收容狗与猫，且严格遵守“先进先出”的原则。在收养该收容所的动物时，收养人只能收养所有动物中“最老”（根据进入收容所的时间长短）的动物，或者，可以挑选猫或狗（同时必须收养此类动物中“最老”的）。换言之，收养人不能自由挑选想收养的对象。请创建适用于这个系统的数据结构，实现各种操作方法，比如enqueue、dequeueAny、dequeueDog和dequeueCat等。允许使用Java内置的LinkedList数据结构。（第 51 页）

解法

这个问题有多种不同的解法。比如，我们可以只维护一个队列。这么做的话，dequeueAny（收养任意一种动物）实现起来很简单，但dequeueDog（收养狗）和dequeueCat（收养猫）就要迭代访问整个队列，才能找到第一只该被收养的狗或猫。这会增加整个解法的复杂度，降低执行效率。

另一种解法简单明了而又高效，只需为狗和猫各自创建一个队列，然后将两者放进名为AnimalQueue的包裹类，并且存储某种形式的时戳，以标记每只动物进入队列（即收容所）的时间。当调用dequeueAny时，查看狗队列和猫队列的首部，并返回“最老”的那一只。

```
public abstract class Animal {
   private int order;
   protected String name;
   public Animal(String n) {
     name = n;
   }

   public void setOrder(int ord) {
     order = ord;
   }

   public int getOrder() {
     return order;
   }

   public boolean isOlderThan(Animal a) {
     return this.order < a.getOrder();
   }
}

public class AnimalQueue {
   LinkedList<Dog> dogs = new LinkedList<Dog>();
   LinkedList<Cat> cats = new LinkedList<Cat>();
   private int order = 0; // 用作时戳

   public void enqueue(Animal a) {
     /* order用作某种形式的时戳，以便比较狗或猫
      * 插入队列的先后顺序 */
     a.setOrder(order);
     order++;

     if (a instanceof Dog) dogs.addLast((Dog) a);
     else if (a instanceof Cat) cats.addLast((Cat)a);
```

```
  }

  public Animal dequeueAny() {
    /* 查看狗和猫的队列的首部，弹出
     * 最旧的值 */
    if (dogs.size() == 0) {
      return dequeueCats();
    } else if (cats.size() == 0) {
      return dequeueDogs();
    }

    Dog dog = dogs.peek();
    Cat cat = cats.peek();
    if (dog.isOlderThan(cat)) {
      return dequeueDogs();
    } else {
      return dequeueCats();
    }
  }
  public Dog dequeueDogs() {
    return dogs.poll();
  }

  public Cat dequeueCats() {
    return cats.poll();
  }
}

public class Dog extends Animal {
  public Dog(String n) {
    super(n);
  }
}

public class Cat extends Animal {
  public Cat(String n) {
    super(n);
  }
}
```

9.4　树与图

4.1　实现一个函数，检查二叉树是否平衡。在这个问题中，平衡树的定义如下：任意一个结点，其两棵子树的高度差不超过 1。（第 54 页）

解法

还算幸运，此题至少明确给出了平衡树的定义：任意一个结点，其两棵子树的高度差不大于1。根据该定义可以得到一种解法，即直接递归访问整棵树，计算每个结点两棵子树的高度。

```
public static int getHeight(TreeNode root) {
   if (root == null) return 0; // 终止条件
   return Math.max(getHeight(root.left),
                   getHeight(root.right)) + 1;
}

public static boolean isBalanced(TreeNode root) {
   if (root == null) return true; // 终止条件

   int heightDiff = getHeight(root.left) - getHeight(root.right);
   if (Math.abs(heightDiff) > 1) {
     return false;
   } else { // 递归
     return isBalanced(root.left) && isBalanced(root.right);
   }
}
```

虽然可行，但效率不太高，这段代码会递归访问每个结点的整棵子树。也就是说，getHeight会被反复调用计算同一个结点的高度。因此，这个算法的时间复杂度为$O(N \log N)$。

我们可以删减部分getHeight调用。

仔细查看上面的方法，你或许会发现，getHeight不仅可以检查高度，还能检查这棵树是否平衡。那么，我们发现子树不平衡时又该怎么做呢？直接返回–1。

改进过的算法会从根结点递归向下检查每棵子树的高度。我们会通过checkHeight方法，以递归方式获取每个结点左右子树的高度。若子树是平衡的，则checkHeight返回该子树的实际高度。若子树不平衡，则checkHeight返回–1。checkHeight会立即中断执行，并返回–1。

下面是该算法的实现代码。

```
public static int checkHeight(TreeNode root) {
   if (root == null) {
     return 0; // 高度为0
   }

   /* 检查左子树是否平衡 */
   int leftHeight = checkHeight(root.left);
   if (leftHeight == -1) {
     return -1; // 不平衡
   }
   /* 检查左子树是否平衡 */
   int rightHeight = checkHeight(root.right);
   if (rightHeight == -1) {
     return -1; // 不平衡
   }

   /* 检查当前结点是否平衡 */
   int heightDiff = leftHeight - rightHeight;
   if (Math.abs(heightDiff) > 1) {
     return -1; // 不平衡
   } else {
     /* 返回高度 */
     return Math.max(leftHeight, rightHeight) + 1;
   }
```

```
}

public static boolean isBalanced(TreeNode root) {
   if (checkHeight(root) == -1) {
     return false;
   } else {
     return true;
   }
}
```

这段代码需要$O(N)$的时间和$O(H)$的空间，其中H为树的高度。

4.2 给定有向图，设计一个算法，找出两个结点之间是否存在一条路径。（第 54 页）

解法

只需通过图的遍历，比如深度优先搜索或广度优先搜索等，就能解决这个问题。我们从两个结点的其中一个出发，在遍历过程中，检查是否找到另一个结点。在这个算法中，访问过的结点都应标记为“已访问”，以免循环和重复访问结点。

下面是广度优先搜索的迭代实现。

```
public enum State {
   Unvisited, Visited, Visiting;
}

public static boolean search(Graph g, Node start, Node end) {
   // 当作队列使用
   LinkedList<Node> q = new LinkedList<Node>();

   for (Node u : g.getNodes()) {
     u.state = State.Unvisited;
   }
   start.state = State.Visiting;
   q.add(start);
   Node u;
   while (!q.isEmpty()) {
     u = q.removeFirst(); // 也即dequeue()
     if (u != null) {
        for (Node v : u.getAdjacent()) {
          if (v.state == State.Unvisited) {
            if (v == end) {
              return true;
            } else {
              v.state = State.Visiting;
              q.add(v);
            }
          }
        }
        u.state = State.Visited;
     }
   }
   return false;
}
```

碰到这类问题时，很有必要跟面试官探讨一下广度优先搜索和深度优先搜索之间的利弊。例如，深度优先搜索实现起来比较简单，因为只需简单的递归即可。广度优先搜索很适合用来查找最短路径，而深度优先搜索在访问邻近结点之前，可能会先深度遍历其中一个邻近结点。

4.3 给定一个有序整数数组，元素各不相同且按升序排列，编写一个算法，创建一棵高度最小的二叉查找树。**（第 54 页）**

解法

要创建一棵高度最小的树，就必须让左右子树的结点数量越接近越好。也就是说，我们要让数组中间的值成为根结点，这么一来，数组左边一半就成为左子树，右边一半成为右子树。

然后，我们继续以类似方式构造整棵树。数组每一区段的中间元素成为子树的根结点，左半部分成为左子树，右半部分成为右子树。

一种实现方式是使用简单的`root.insertNode(int v)`方法，从根结点开始，以递归方式将值v插入树中。这么做的确能构造最小高度的树，但效率并不是太高。每次插入操作都要遍历整棵树，时间开销为$O(N \log N)$。

另一种做法是以递归方式运用`createMinimalBST`方法，从而消除部分多余的遍历操作。这个方法会传入数组的一个区段，并返回最小树的根结点。

该算法简述如下。

(1) 将数组中间位置的元素插入树中。

(2) 将数组左半边元素插入左子树。

(3) 将数组右半边元素插入右子树。

(4) 递归处理。

下面是该算法的实现代码。

```
TreeNode createMinimalBST(int arr[], int start, int end) {
   if (end < start) {
     return null;
   }
   int mid = (start + end) / 2;
   TreeNode n = new TreeNode(arr[mid]);
   n.left = createMinimalBST(arr, start, mid - 1);
   n.right = createMinimalBST(arr, mid + 1, end);
   return n;
}

TreeNode createMinimalBST(int array[]) {
   return createMinimalBST(array, 0, array.length - 1);
}
```

尽管这段代码看起来不是特别复杂，但在编写过程中很容易犯了差一错误（off-by-one）。对这部分代码，务必进行详尽测试。

4.4　给定一棵二叉树，设计一个算法，创建含有某一深度上所有结点的链表（比如，若一棵树的深度为D，则会创建出D个链表）。**（第 54 页）**

解法

乍一看，你可能认为这个问题需要一层一层逐一遍历，但其实并无必要。你可以用任意方式遍历整棵树，只要记住结点位于哪一层即可。

我们可以将前序遍历算法稍作修改，将`level + 1`传入下一个递归调用。下面是使用深度优先搜索的实现代码。

```
void createLevelLinkedList(TreeNode root,
    ArrayList<LinkedList<TreeNode>> lists, int level) {
  if (root == null) return; // 终止条件

  LinkedList<TreeNode> list = null;
  if (lists.size() == level) { // 该层不在链表中
    list = new LinkedList<TreeNode>();
    /* 以中序遍历所有层级，因此，若这是第一次
     * 访问第i层，则表示我们已访问过第0到i-1层。
     * 因此，我们可以安全地将这一层加到链表
     * 末端。 */
    lists.add(list);
  } else {
    list = lists.get(level);
  }
  list.add(root);
  createLevelLinkedList(root.left, lists, level + 1);
  createLevelLinkedList(root.right, lists, level + 1);
}

ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
    TreeNode root) {
  ArrayList<LinkedList<TreeNode>> lists =
    new ArrayList<LinkedList<TreeNode>>();
  createLevelLinkedList(root, lists, 0);
  return lists;
}
```

另一种做法是对广度优先搜索稍加修改，即从根结点开始迭代，然后第2层，第3层，等等。

处于第i层时，则表明我们已访问过第$i-1$层的所有结点。也就是说，要得到i层的结点，只需直接查看$i-1$层结点的所有子结点即可。

下面是该算法的实现代码。

```
ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
    TreeNode root) {
  ArrayList<LinkedList<TreeNode>> result =
    new ArrayList<LinkedList<TreeNode>>();
  /* 访问根结点 */
  LinkedList<TreeNode> current = new LinkedList<TreeNode>();
  if (root != null) {
    current.add(root);
```

```
  }

  while (current.size() > 0) {
    result.add(current); // 加入上一层
    LinkedList<TreeNode> parents = current; // 转到下一层
    current = new LinkedList<TreeNode>();
    for (TreeNode parent : parents) {
      /* 访问子结点 */
      if (parent.left != null) {
        current.add(parent.left);
      }
      if (parent.right != null) {
        current.add(parent.right);
      }
    }
  }
  return result;
}
```

你可能会问，这两种解法哪一种效率更高？两者的时间复杂度皆为$O(N)$，那么空间效率呢？乍一看，我们可能会以为第二种解法的空间效率更高。

在某种意义上，这么说也对。第一种解法会用到$O(\log N)$次递归调用（在平衡树中），每次调用都会在栈里增加一级。第二种解法采用迭代遍历法，不需要这部分额外空间。

不过，两种解法都要返回$O(N)$数据，因此，递归实现所需的额外$O(\log N)$空间，跟必须传回的$O(N)$数据相比，并不算多。虽然第一种解法确实使用了较多的空间，但从大O记法的角度来看，两者效率是一样的。

4.5　实现一个函数，检查一棵二叉树是否为二叉查找树。（第 54 页）

解法

此题有两种不同的解法。第一种是利用中序遍历，第二种则建立在`left <= current < right`这项特性之上。

解法 1：中序遍历

看到此题，闪过的第一个想法可能是中序遍历，将所有元素复制到数组中，然后检查该数组是否有序。这种解法要多用一点内存，大部分情况下都没问题。

唯一的问题在于，它无法正确处理树中的重复值。例如，该算法无法区分下面这两棵树（其中一棵是无效的），因为两者的中序遍历结果相同。

```
Valid BST [20.left = 20]
Invalid BST [20.right = 20]
```

不过，要是假定这棵树不得包含重复值，那么这种做法还是行之有效的。该方法的伪码大致如下：

```
public static int index = 0;
public static void copyBST(TreeNode root, int[] array) {
  if (root == null) return;
```

```
  copyBST(root.left, array);
  array[index] = root.data;
  index++;
  copyBST(root.right, array);
}

public static boolean checkBST(TreeNode root) {
  int[] array = new int[root.size];
  copyBST(root, array);
  for (int i = 1; i < array.length; i++) {
    if (array[i] <= array[i - 1]) return false;
  }
  return true;
}
```

注意，这里必须记录数组在逻辑上的“尾部”，用它来分配空间以储存所有元素。

仔细审视这个解法，我们就会发现代码中的数组实无必要。除了用来比较某个元素和前一个元素，别无他用。那么，为什么不在进行比较时，直接记下最后的元素？

下面是该算法的实现代码。

```
public static int last_printed = Integer.MIN_VALUE;
public static boolean checkBST(TreeNode n) {
  if (n == null) return true;

    // 递归检查左子树
    if (!checkBST(n.left)) return false;

    // 检查当前结点
    if (n.data <= last_printed) return false;
    last_printed = n.data;

    // 递归检查右子树
    if (!checkBST(n.right)) return false;

    return true; // 全部检查完毕
  }
```

要是不喜欢使用静态变量，可以稍作修改，使用包裹类存放这个整数值，如下所示：

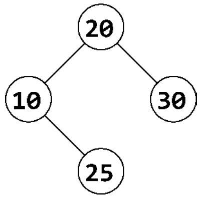
```
class WrapInt {
  public int value;
}
```

或者，若用C++或其他支持按引用传值的语言实现，就可以这么做。

解法 2：最小/最大法

第二种解法利用的是二叉查找树的定义。

一棵什么样的树才成其为二叉查找树？我们知道这棵树必须满足以下条件：对于每个结点，`left.data <= current.data < right.data`，但是这样还不够。试看下面这棵小树：

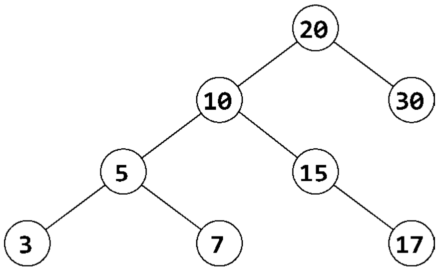


尽管每个结点都比左子结点大，比右子结点小，但这显然不是一棵二叉查找树，其中25的位置不对。

更准确地说，成为二叉查找树的条件是：所有左边的结点必须小于或等于当前结点，而当前结点必须小于所有右边的结点。

利用这一点，我们可以通过自上而下传递最小和最大值来解决这个问题。在迭代遍历整个树的过程中，我们会用逐渐变窄的范围来检查各个结点。

以下面这棵树为例：



首先，从`(min = INT_MIN, max = INT_MAX)`这个范围开始，根结点显然落在其中。然后处理左子树，检查这些结点是否落在`(min = INT_MIN, max = 20)`范围内。然后再处理（值为10的结点）右子树，检查结点是否落在`(min = 20, max = INT_MAX)`范围内。

然后，继续依此遍历整棵树。进入左子树时，更新`max`。进入右子树时，更新`min`。只要有任一结点不能通过检查，则停止并返回`false`。

这种解法的时间复杂度为$O(N)$，其中N为整棵树的结点数。我们可以证明这已经是最佳做法，因为任何算法都必须访问全部N个结点。

因为用了递归，对于平衡树，空间复杂度为$O(\log N)$。在调用栈上，共有$O(\log N)$个递归调用，因为递归的深度最大会到这棵树的深度。

该解法的递归实现代码如下：

```
boolean checkBST(TreeNode n) {
   return checkBST(n, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

boolean checkBST(TreeNode n, int min, int max) {
   if (n == null) {
     return true;
   }
```

```
  if (n.data < min || n.data >= max) {
    return false;
  }

  if (!checkBST(n.left, min, n.data) ||
     !checkBST(n.right, n.data, max)) {
    return false;
  }
  return true;
}
```

记住，在递归算法中，一定要确定终止条件以及结点为空的情况得到妥善处理。

4.6　设计一个算法，找出二叉查找树中指定结点的“下一个”结点（也即中序后继）。可以假定每个结点都含有指向父结点的连接。（第 54 页）

解法

回想一下中序遍历，它会遍历左子树，然后是当前结点，接着是右子树。要解决这个问题，必须非常小心，想想具体是怎么回事。

假定我们有一个假想的结点。我们知道访问顺序为左子树，当前结点，然后是右子树。显然，下一个结点应该位于右边。

不过，到底是右子树的哪个结点呢？如果中序遍历右子树，那它就会是接下来第一个被访问的结点，也就是说，它应该是右子树最左边的结点。够简单吧！

但是，若这个结点没有右子树，又该怎么办？这种情况就有点棘手了。

若结点n没有右子树，那就表示已遍访n的子树。我们必须回到n的父结点，记作q。

若n在q的左边，那么，下一个我们应该访问的结点就是q（中序遍历，`left -> current -> right`）。

若n在q的右边，则表示已遍访q的子树。我们需要从q往上访问，直至找到我们还未完全遍访过的结点x。怎么才能知道还未完全遍历结点x呢？之前从左结点访问至其父结点时，就已碰到了这种情况。左结点已完全遍历，但其父结点尚未完全遍历。

伪码如下：

```
Node inorderSucc(Node n) {
  if (n has a right subtree) {
    return leftmost child of right subtree
  } else {
    while (n is a right child of n.parent) {
      n = n.parent; // 往上
    }
    return n.parent; // 父结点还未遍历
  }
}
```

且慢，如果一路往上遍访这棵树都没发现左结点呢？只有当我们遇到中序遍历的最末端时，才会出现这种情况。也就是说，如果我们已位于树的最右边，那就不会再有中序后继，此时该返回`null`。

下面是该算法的实现代码（已正确处理结点为空的情况）。

```
public TreeNode inorderSucc(TreeNode n) {
   if (n == null) return null;

   /* 找到右子结点，则返回右子树里
    * 最左边的结点 */
   if (n.right != null)
     return leftMostChild(n.right);
   } else {
     TreeNode q = n;
     TreeNode x = q.parent;
     // 向上直至位于左边而不是右边
     while (x != null && x.left != q) {
        q = x;
        x = x.parent;
     }
     return x;
   }
}

public TreeNode leftMostChild(TreeNode n) {
   if (n == null) {
     return null;
   }
   while (n.left != null) {
     n = n.left;
   }
   return n;
}
```

这不是世上最复杂的算法问题，但要写出完美无瑕的代码却有难度。面对这类问题，比较实用的做法是用伪码勾勒大纲，仔细描绘各种不同的情况。

4.7　设计并实现一个算法，找出二叉树中某两个结点的第一个共同祖先。不得将额外的结点储存在另外的数据结构中。注意：这不一定是二叉查找树。**（第 54 页）**

解法

如果是二叉查找树，我们可以修改`find`操作，用来查找这两个结点，看看路径在哪里开始分叉。可惜，这不是二叉查找树，因此必须另觅他法。

下面假定我们要找出结点`p`和`q`的共同祖先。在此先要问个问题，这棵树的结点是否包含指向父结点的连接。

解法 1：包含指向父结点的连接

如果每个结点都包含指向父结点的连接，我们就可以向上追踪`p`和`q`的路径，直至两者相交。不过，这么做可能不符合题目的若干假设，因为它需要满足以下两个条件之一：(1) 可将结点标记为`isVisited`；(2) 可用另外的数据结构如散列表储存一些数据。

解法 2：不包含指向父结点的连接

另一种做法是，顺着一条p和q都在同一边的链子，也就是说，若p和q都在某结点的左边，就到左子树中查找共同祖先。若都在右边，则在右子树中查找共同祖先。要是p和q不在同一边，那就表示已经找到第一个共同祖先。

这种做法的实现代码如下。

```
/* 若p为root的子孙，则返回true */
boolean covers(TreeNode root, TreeNode p) {
   if (root == null) return false;
   if (root == p) return true;
   return covers(root.left, p) || covers(root.right, p);
}

TreeNode commonAncestorHelper(TreeNode root, TreeNode p,
                              TreeNode q) {
   if (root == null) return null;
   if (root == p || root == q) return root;

   boolean is_p_on_left = covers(root.left, p);
   boolean is_q_on_left = covers(root.left, q);

   /* 若p和q不在同一边，则返回root */
   if (is_p_on_left != is_q_on_left) return root;

   /* 否则就是在同一边，遍访那一边 */
   TreeNode child_side = is_p_on_left ? root.left : root.right;
   return commonAncestorHelper(child_side, p, q);
}

TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
   if (!covers(root, p) || !covers(root, q)) { // 错误检查
     return null;
   }
   return commonAncestorHelper(root, p, q);
}
```

这个算法在平衡树上的运行时间为$O(n)$。这是因为第一次调用时，covers会在$2n$个结点上调用（左边n个结点，右边n个结点）。接着，该算法会访问左子树或右子树，此时covers会在$2n/2$个结点上调用，然后是$2n/4$，依此类推。最终的运行时间为$O(n)$。

至此，就渐近式运行时间（asymptotic runtime）来看，可以确定没有更优解了，因为必须遍访这棵树的每一个结点才行。不过，或许我们还能减小常数倍的值。

解法 3：最优化

尽管解法2在运行时间上已经做到最优，还是可以看出部分低效的操作。特别是，covers会搜索root下的所有结点以查找p和q，包括每棵子树中的结点（root.left和root.right）。然后，它会选择那些子树中的一棵，搜遍它的所有结点。每棵子树都会被一再地反复搜索。

你可能会觉察到，只需搜索一遍整棵树，就能找到p和q。然后，就可以“往上冒泡”在栈里找到先前的结点。基本逻辑与上一种解法相同。

使用函数commonAncestor(TreeNode root, TreeNode p, TreeNode q)递归访问整棵树，这个函数的返回值如下：

- 返回p，若root的子树含有p（而非q）；
- 返回q，若root的子树含有q（而非p）；
- 返回null，若p和q都不在root的子树中；
- 否则，返回p和q的共同祖先。

在最后一种情况下，要找到p和q的共同祖先比较简单。当commonAncestor(n.left, p, q)和commonAncestor(n.right, p, q)都返回非空的值时（意即p和q位于不同的子树中），则n即为共同祖先。

下面的代码提供了初步的解法，不过其中有个bug。试着找找看。

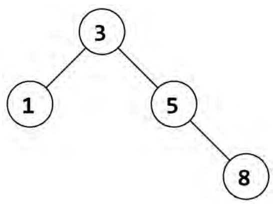
```
/* 下面的代码有个bug */
TreeNode commonAncestorBad(TreeNode root, TreeNode p, TreeNode q) {
   if (root == null) {
     return null;
   }
   if (root == p && root == q) {
     return root;
   }

   TreeNode x = commonAncestorBad(root.left, p, q);
   if (x != null && x != p && x != q) { // 已经找到父系结点
     return x;
   }

   TreeNode y = commonAncestorBad(root.right, p, q);
   if (y != null && y != p && y != q) { // 已经找到父系结点
     return y;
   }

   if (x != null && y != null) { // 在不同子树里找到p和q
     return root; // 这是共同祖先
   } else if (root == p || root == q) {
     return root;
   } else {
     /* x或y有一个非空，则返回非空的那个值 */
     return x == null ? y : x;
   }
}
```

假如有个结点不在这棵树中，这段代码就会出问题。例如，请看下面这棵树：



假设我们调用`commonAncestor(node 3, node 5, node 7)`。当然，结点7并不存在，而这正是问题的源头。调用序列如下：

```
commonAncestor(node 3, node 5, node 7)              // --> 5
  calls commonAncestor(node 1, node 5, node 7)      // --> null
  calls commonAncestor(node 5, node 5, node 7)      // --> 5
    calls commonAncestor(node 8, node 5, node 7)    // --> null
```

换句话说，对右子树调用`commonAncestor`时，前面的代码会返回结点5，这也符合代码本意。问题在于查找p和q的共同祖先时，调用函数无法区分下面两种情况。

- 情况1：p是q的子结点（或相反，q为p的子结点）。
- 情况2：p在这棵树中，而q不在这棵树中（或者相反）。

不论哪种情况，`commonAncestor`都将返回p。对于情况1，这是正确的返回值，而对于情况2，返回值应该为`null`。

我们需要设法区分这两种情况，这也是以下代码所做的。这段代码的做法是返回两个值：结点自身，以及指示这个结点是否确为共同祖先的标记。

```
public static class Result {
  public TreeNode node;
  public boolean isAncestor;
  public Result(TreeNode n, boolean isAnc) {
    node = n;
    isAncestor = isAnc;
  }
}

Result commonAncestorHelper(TreeNode root, TreeNode p, TreeNode q){
  if (root == null) {
    return new Result(null, false);
  }
  if (root == p && root == q) {
    return new Result(root, true);
  }

  Result rx = commonAncestorHelper(root.left, p, q);
  if (rx.isAncestor) { // 找到共同祖先
    return rx;
  }

  Result ry = commonAncestorHelper(root.right, p, q);
  if (ry.isAncestor) { // 找到共同祖先
    return ry;
  }

  if (rx.node != null && ry.node != null) {
    return new Result(root, true); // 这是共同祖先
  } else if (root == p || root == q) {
    /* 若我们当前位于p或q，并发现其中一个结点
     * 位于子树中，那么这真的就是一个共同祖先，
     * 标记应该设为true。 */
```

```
    boolean isAncestor = rx.node != null || ry.node != null ?
                         true : false;
    return new Result(root, isAncestor);
  } else {
    return new Result(rx.node!=null ? rx.node : ry.node, false);
  }
}

TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  Result r = commonAncestorHelper(root, p, q);
  if (r.isAncestor) {
    return r.node;
  }
  return null;
}
```

当然，由于这个问题只会在p或q并不属于这棵树的情况下出现，另一种避免bug的做法是先搜遍整棵树，以确保两个结点都在树中。

4.8　你有两棵非常大的二叉树：T1，有几百万个结点；T2，有几百个结点。设计一个算法，判断T2 是否为T1 的子树。

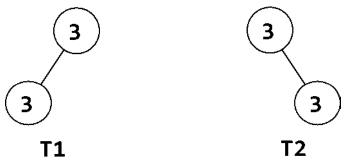
如果T1 有这么一个结点n，其子树与T2 一模一样，则T2 为T1 的子树。也就是说，从结点n 处把树砍断，得到的树与T2 完全相同。（第 54 页）

解法

碰到类似的问题，不妨假设只有少量的数据，以此为基础解决问题。这么做很有用，可以借此找出可行的基本解法。

在规模较小且较简单的问题中，我们可以创建一个字符串，表示中序和前序遍历。若T2前序遍历是T1前序遍历的子串，并且T2中序遍历是T1中序遍历的子串，则T2为T1的子树。利用后缀树可以在线性时间内检查是否为子串，因此就最差情况的时间复杂度而言，这个算法是相当高效的。

注意，我们需要在字符串中插入特殊字符，表示左结点或右结点为NULL的情况。否则，我们就无法区分以下两种情况：



尽管这两棵树不同，但两者的中序和前序遍历完全一样。

```
T1，中序：3, 3
T1，前序：3, 3
T2，中序：3, 3
T2，前序：3, 3
```

不过，要是标记出NULL值，就能区分这两棵树：

```
T1，中序：0, 3, 0, 3, 0
T1，前序：3, 3, 0, 0, 0
T2，中序：0, 3, 0, 3, 0
T2，前序：3, 0, 3, 0, 0
```

对于简单的情形，这种解法还算不错，但是我们真正要面对的问题涉及的数据量要大得多。鉴于该问题指定的约束条件，创建两棵树的副本可能要占用太多的内存。

另一种解法

另一种解法是搜遍较大的那棵树T1。每当T1的某个结点与T2的根结点匹配时，就调用`treeMatch`。`treeMatch`方法会比较两棵子树，检查两者是否相同。

分析运行时间有点复杂，粗略一看的答案可能是$O(nm)$，其中n为T1的结点数，m为T2的结点数。虽然在技术上这个答案是正确的，但稍微再想想就能得到更靠谱的答案。

我们不必对T2的每个结点调用`treeMatch`，而是会调用k次，其中k为T2根结点在T1中出现的次数。因此运行时间接近$O(n + km)$。

其实，即使这样运行时间也有所夸大。即使根结点相同，一旦发现T1和T2有结点不同，我们就会退出`treeMatch`。因此，每次调用`treeMatch`，也不见得都会查看m个结点。

下面是该算法的实现代码。

```java
boolean containsTree(TreeNode t1, TreeNode t2) {
   if (t2 == null) { // 空树一定是子树
     return true;
   }
   return subTree(t1, t2);
}

boolean subTree(TreeNode r1, TreeNode r2) {
   if (r1 == null) {
     return false; // 大的树已经空了，还未找到子树
   }
   if (r1.data == r2.data) {
     if (matchTree(r1,r2)) return true;
   }
   return (subTree(r1.left, r2) || subTree(r1.right, r2));
}

boolean matchTree(TreeNode r1, TreeNode r2) {
   if (r2 == null && r1 == null) // 若两者都空
     return true; // 子树中已无结点

   // 若其中之一为空，但并不同时为空
   if (r1 == null || r2 == null) {
     return false;
   }

   if (r1.data != r2.data)
     return false;  // 结点数据不匹配
   return (matchTree(r1.left, r2.left) &&
        matchTree(r1.right, r2.right));
```

```
31    }
32 }
```

什么情况下用简单解法比较好，什么时候另一种解法比较好呢？这个问题值得跟面试官好好讨论一番，下面是几点注意事项。

(1) 简单解法会占用$O(n+m)$内存，而另一种解法则占用$O(\log(n)+\log(m))$内存。记住：要求可扩展性时，内存使用多寡关系重大。

(2) 简单解法的时间复杂度为$O(n+m)$，另一种解法在最差情况下的执行时间为$O(nm)$。话说回来，只看最差情况的时间复杂度可能会造成误导，我们需要做进一步观察。

(3) 如前所述，比较准的运行时间为$O(n+km)$，其中k为T2根结点在T1中出现的次数。假设T1和T2的结点数据为0和p之间的随机数，则k值大约为n/p，为什么？因为T1有n个结点，每个结点有$1/p$的几率与T2根结点相同，因此，T1中大约有n/p个结点等于T2根结点（`T2.root`）。举个例子，假设`p = 1000`，`n = 1 000 000`且`m = 100`。我们需要检查的结点数量大致为1 100 000（`1 100 000 = 1 000 000 + 100*1 000 000/1000`）。

（4）借助更复杂的数学运算和假设，就能得到更准确的运行时间。在第3点中，我们假设调用`treeMatch`时将遍历T2的全部m个结点。然而，更有可能出现的情况是，我们很早就发现两棵树有不同的结点，然后提早就退出了这个函数。

总的来说，在空间使用上，另一种解法显然比较好，在时间复杂度上，也可能比简单解法更优。一切都取决于你做出哪些假设，以及要不要考虑牺牲最差情况的运行时间，来减少平均情况的运行时间。这一点非常值得向面试官提出并讨论。

4.9 给定一棵二叉树，其中每个结点都含有一个数值。设计一个算法，打印结点数值总和等于某个给定值的所有路径。注意，路径不一定非得从二叉树的根结点或叶结点开始或结束。（第54页）

解法

下面我们运用简化推广法来解题。

部分1：简化——假设路径必须从根结点开始，但可以在任意结点结束，怎么解决？

在这种情况下，问题就会变得容易很多。

我们可以从根结点开始，向左向右访问子结点，计算每条路径上到当前结点为止的数值总和，若与给定值相同则打印当前路径。注意，就算找到总和，仍要继续访问这条路径。为什么？因为这条路径可能继续往下经过`a + 1`结点和`a - 1`结点（或其他数值总和为0的结点序列），完整路径的总和仍然等于`sum`。

例如，若`sum = 5`，可能会得到以下路径：

- `p = {2, 3}`
- `q = {2, 3, -4, -2, 6}`

如果找到`2 + 3`就停下来，我们就会错过第二条路径，还可能错过其他路径。因此，我们必须继续往下查找所有可能的路径。

部分 2：推广——路径可从任意结点开始。

现在，如果路径可从任意结点开始，该怎么办？在这种情况下，我们可以稍作调整。对于每个结点，我们都会向“上”检查是否得到相符的总和。也就是说，我们不再要求“从这个结点开始是否会有总和为给定值的路径”，而是关注“这个结点是否为总和为给定值的某条路径的末端”。

递归访问每个结点n时，我们会将root到n的完整路径传入该函数。随后，这个函数会以相反的顺序，从n到root，将路径上的结点值加起来。当每条子路径的总和等于sum时，就打印这条路径。

```
public void findSum(TreeNode node, int sum, int[] path, int level) {
   if (node == null) {
     return;
   }

   /* 将当前结点插入路径 */
   path[level] = node.data;

   /* 查找以此为终点且总和为sum的路径 */
   int t = 0;
   for (int i = level; i >= 0; i--){
     t += path[i];
     if (t == sum) {
        print(path, i, level);
     }
   }

   /* 查找此结点之下的结点 */
   findSum(node.left, sum, path, level + 1);
   findSum(node.right, sum, path, level + 1);

   /* 从路径中移除当前结点。严格来说并不一定要这么做，
    * 直接忽略这个值即可，但这么做是个好习惯 */
   path[level] = Integer.MIN_VALUE;
}

public void findSum(TreeNode node, int sum) {
   int depth = depth(node);
   int[] path = new int[depth];
   findSum(node, sum, path, 0);
}

public static void print(int[] path, int start, int end) {
   for (int i = start; i <= end; i++) {
     System.out.print(path[i] + “ “);
   }
   System.out.println();
}

public int depth(TreeNode node) {
   if (node == null) {
     return 0;
   } else {
```

```
    return 1 + Math.max(depth(node.left), depth(node.right));
  }
}
```

那么，这个算法的时间复杂度如何（假设是棵平衡二叉树）？如果结点在r层，那么就需要r份量的工作（向“上”检查结点的步骤）。我们可以猜测时间复杂度为$O(n \log(n))$，因为总共有n个结点，平均下来，每一步需要$\log(n)$的工作量。

如果这么分析，你还是看不大明白，我们也可以用严格的数学推导来说明。注意，在r层上有2^r个结点。

$$1 * 2^1 + 2 * 2^2 + 3 * 2^3 + 4 * 2^4 + \dots d * 2^d$$
$$= \text{sum}(r * 2^r, r \text{ from 0 to depth})$$
$$= 2 * (d - 1) * 2^d + 2$$
$$n = 2^d$$
$$d = \log(n)$$

注意，$2^{\log(x)} = x$，因此，

$$O(2 * (\log(n) - 1) * 2^{\log(n)} + 2)$$
$$= O(2 (\log n - 1) * n)$$
$$= O(n \log(n))$$

按照同样的逻辑，可以推导出算法的空间复杂度为$O(\log(n))$，因为该算法会递归$O(\log n)$次，而在递归调用中参数path只分配一次空间（大小为$O(\log n)$）。

9.5 位操作

5.1 给定两个 32 位的整数*N*与*M*，以及表示比特位置的*i*与*j*。编写一个方法，将*M*插入*N*，使得*M*从*N*的第*j*位开始，到第*i*位结束。假定从*j*位到*i*位足以容纳*M*，也即若*M*=10011，那么*j*和*i*之间至少可容纳 5 个位。例如，不可能出现*j* = 3 和*i* = 2 的情况，因为第 3 位和第 2 位之间放不下*M*。

示例输入：N = 10000000000, M = 10011, *i* = 2, *j* = 6 输出：N = 10001001100（第 56 页）

解法

这个问题的解决可分为三大步骤。

(1) 将N中从j到i之间的位清零。

(2) 对M执行移位操作，与j和i之间的位对齐。

(3) 合并M与N。

其中步骤1最为棘手。如何将N中的那些位清零呢？我们可以利用掩码来清零。除j到i之间的位为0外，这个掩码的其余位均为1。我们会先创建掩码的左半部分，然后是右半部分，最终得到整个掩码。

```
int updateBits(int n, int m, int i, int j) {
  /* 创建掩码，用来清除n中i到j的位
  /* 示例：i = 2, j = 4。掩码为11100011。
```

```
   * 为简单起见，本例掩码只有8位
   */
  int allOnes = ~0; // 等同于一连串的1

  // 在位置j之前的位均为1，其余为0，left = 11100000
  int left = allOnes << (j + 1);

  // 在位置i之后的位均为1，right = 00000011
  int right = ((1 << i) - 1);

  // 除i到j的位为0，其余位均为1。mask = 11100011
  int mask = left | right;

  /* 清除位置j到i的位，然后将m放进去 */
  int n_cleared = n & mask; // 清除j到i的位
  int m_shifted = m << i; // 将m移至相应的位置

  return n_cleared | m_shifted; // 对两者执行位或操作，搞定！
}
```

解决这类问题时（包括许多位操作问题），务必切实充分地对代码进行测试。否则，一不小心就容易犯下差一错误。

5.2　给定一个介于 0 和 1 之间的实数（如 0.72），类型为double，打印它的二进制表示。如果该数字无法精确地用 32 位以内的二进制表示，则打印“ERROR”。（第 56 页）

解法

注意，为表示清晰起见，这里分别用x_2和x_{10}来指示x是二进制还是十进制。

首先，我们要弄清楚非整型的数字用二进制表示是什么样的。与十进制数相仿，二进制数0.101_2表示如下：

$$0.101_2 = 1 * (1/2^1) + 0 * (1/2^2) + 1 * (1/2^3)$$

为了打印小数部分，我们可以将这个数乘以2，检查$2n$是否大于或等于1。这实质上等同于“移动”小数部分，也即：

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * (1/2^0) + 0 * (1/2^1) + 1 * (1/2^2) \\ &= 1.01_2 \end{aligned}$$

若`r >= 1`，可知n的小数点后面正好有个1。不断重复上述步骤，我们可以检查每个数位。

```
public static String printBinary(double num) {
  if (num >= 1 || num <= 0) {
    return “ERROR”;
  }

  StringBuilder binary = new StringBuilder();
  binary.append(“.”);
  while (num > 0) {
    /* 设定长度上限：32个字符 */
```

```
        if (binary.length() >= 32) {
            return “ERROR”;
        }

        double r = num * 2;
        if (r >= 1) {
            binary.append(1);
            num = r - 1;
        } else {
            binary.append(0);
            num = r;
        }
    }
    return binary.toString();
}
```

上面的做法是将数字乘以2，然后与1进行比较，此外我们还可以将这个数与0.5比较，然后与0.25比较，依此类推。下面的代码示范了这一做法。

```
public static String printBinary2(double num) {
    if (num >= 1 || num <= 0) {
      return “ERROR”;
    }

    StringBuilder binary = new StringBuilder();
    double frac = 0.5;
    binary.append(“.”);
    while (num > 0) {
      /* 设定长度上限：32个字符 */
      if (binary.length() > 32) {
         return “ERROR”;
      }
      if (num >= frac) {
         binary.append(1);
         num -= frac;
      } else {
         binary.append(0);
      }
      frac /= 2;
    }
    return binary.toString();
}
```

这两种做法都很不错；具体怎么做，就看你个人觉得哪种做法更自然。

不论采用哪种方式，对于这类问题，一定要准备好详尽的测试用例，并在面试中切实进行测试。

5.3　给定一个正整数，找出与其二进制表示中 1 的个数相同、且大小最接近的那两个数（一个略大，一个略小）。（第 56 页）

解法

这个问题有多种解法，包括蛮力法、位操作以及巧妙运用算术。注意，运用算术法建立在位

操作的解法之上。在介绍算术方法之前，你应该先学会位操作的解法。

1. 蛮力法

简单的做法就是直接使用蛮力：在n的二进制表示中，数出1的个数，然后增加或减小，直至找到1的个数相同的数字。简单吧，但也没什么意思。还有没有更优的做法呢？当然有！

下面先从getNext的代码开始，然后是getPrev。

2. 位操作法：取得后一个较大的数

要是你还在考虑后一个数应该是什么样的，不妨作如下观察。以数字13 948为例，二进制表示如下：

1	1	0	1	1	0	0	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

我们想让这个数大一点（但又不会太大），同时1的个数又要保持想不变。

你会发现：给定一个数n和两个位的位置i和j，假设将位i从1翻转为0，位j从0翻转成1。若$i > j$，n就会减小；若$i < j$，则n就会变大。

继而得到以下几点。

(1) 若将某个0翻转成1，就必须将某个1翻转为0。

(2) 进行位翻转时，如果0变1的位处于1变0的位的左边，这个数字就会变大。

(3) 我们想让这个数变大，但又不致太大。因此，必须翻转最右边的0，且它的右边必须还有个1。

换句话说，我们要翻转最右边但非拖尾的0。用上面的例子来说，拖尾0位于第0到第1个位置。因此，最右边但不是拖尾的0处在位置7。我们把这个位置记作p。

- 步骤1：翻转最右边、非拖尾的0

1	1	0	1	1	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

将位置7翻转后，n就会变大。但是，现在n中的1多了一个，0少了一个。我们还需尽量缩小数值，同时记得满足要求。

缩小数值时，可以重新排列位p右方的那些位，其中，0放到左边，1放到右边。在重新排列的过程中，还要将其中一个1改为0。

有种相对简单的做法是，数出p右方有几个1，将位置0到位置p的所有位清零，然后回填c1-1个1。假设c1为p右方1的个数，c0为p右方0的个数。

下面举例说明这些操作。

- 步骤2：将p右方的所有位清零，由步骤1可知，c0 = 2，c1 = 5，p = 7

1	1	0	1	1	0	1	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

为了将这些位清零，需要创建一个掩码，前面是一连串的1，后面跟着p个0，做法如下：

```
a = 1 << p;    // 除位p为1外，其余位均为0
b = a - 1;     // 前面全为0，后面跟p个1
mask = ~b;     // 前面全为1，后面跟p个0
n = n & mask; // 将右边p个位清零
```

或者，更简洁的做法是：

```
n &= ~((1 << p) - 1);
```

- 步骤3：回填`c1 - 1`个1

1	1	0	1	1	0	1	0	0	0	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

要在p右边插入`c1 - 1`个1，做法如下：

```
a = 1 << (c1 - 1); // 位c1 - 1为1，其余位均为0
b = a - 1;         // 位0到位c1 - 1的位为1，其余位均为0
n = n | b;         // 在位0到位c1 - 1处插入1
```

或者，更简洁一点：

```
n |= (1 << (c1 - 1)) - 1;
```

至此，我们得到大于n的数字中，1的个数与n的相同的最小数字。

`getNext`的实现代码如下：

```
public int getNext(int n) {
   /* 计算c0和c1 */
   int c = n;
   int c0 = 0;
   int c1 = 0;
   while (((c & 1) == 0) && (c != 0)) {
     c0++;
     c >>= 1;
   }

   while ((c & 1) == 1) {
     c1++;
     c >>= 1;
   }

   /* 错误：若n == 11..1100...00，那么就没有更大的数字，
    * 且1的个数相同 */
   if (c0 + c1 == 31 || c0 + c1 == 0) {
     return -1;
   }

   int p = c0 + c1; // 最右边、非拖尾0的位置

   n |= (1 << p); // 翻转最右边、非拖尾0
   n &= ~((1 << p) - 1); // 将p右方的所有位清零
   n |= (1 << (c1 - 1)) - 1; // 在右方插入(c1-1)个1
```

```
  return n;
}
```

3. 位操作法：获取前一个较小的数

getPrev的实现方法与getNext的非常相似。

(1) 计算c0和c1。注意c1是拖尾1的个数，而c0为紧邻拖尾1的左方一连串0的个数。

(2) 将最右边、非拖尾1变为0，其位置为p = c1 + c0。

(3) 将位p右边的所有位清零。

(4) 在紧邻位置p的右方，插入c1 + 1个1。

注意，步骤2将位p清零，而步骤3将位0到位p-1清零，我们可以将这两步合并。

下面举例说明各个步骤。

- 步骤1：初始数字，p = 7，c1 = 2，c0 = 5

1	0	0	1	1	1	1	0	0	0	0	0	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

- 步骤2、3：将位0到位p清零

1	0	0	1	1	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

具体做法如下所示：

```
int a = ~0;             // 所有位置1
int b = a << (p + 1);   // 位p左方的所有位为1，后跟p+1个0
n &= b;                 // 将位0到位p清零
```

- 步骤4：在紧邻位置p的右方，插入c1 + 1个1

1	0	0	1	1	1	0	1	1	1	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

注意，p = c1 + c0，因此(c1 + 1)个1的后面会跟(c0 - 1)个0。

```
int a = 1 << (c1 + 1); // 位(c1 + 1)为1，其余位均为0
int b = a - 1;         // 前面为0，后面跟c1 + 1个1
int c = b << (c0 - 1); // c1+1个1，后面跟c0-1个0
n |= c;
```

getPrev的实现代码如下所示。

```
int getPrev(int n) {
  int temp = n;
  int c0 = 0;
  int c1 = 0;
  while (temp & 1 == 1) {
    c1++;
    temp >>= 1;
```

```
    }

    if (temp == 0) return -1;

    while (((temp & 1) == 0) && (temp != 0)) {
      c0++;
      temp >>= 1;
    }

    int p = c0 + c1; // 最右边、非拖尾1的位置
    n &= ((~0) << (p + 1)); // 将位0到位p清零

    int mask = (1 << (c1 + 1)) - 1; // (c1+1)个1
    n |= mask << (c0 - 1);

    return n;
}
```

4. 算术解法：获取后一个数

如果`c0`是拖尾0的个数，`c1`是拖尾0左方全为1的位的个数，而且`p = c0 + c1`，于是我们就可以将前面的解法表述如下。

(1) 将位p置1。

(2) 将位0到位p清零。

(3) 将位0到位`c1 - 1`置1。

步骤1、2有一种快速做法，将拖尾0置为1（得到p个拖尾1），然后再加1。加1后，所有拖尾1都会翻转，最终位p变为1，后面跟p个0。我们可以用算术方法完成这些步骤。

```
n += 2^c0 - 1;   // 将拖尾0置1，得到p个拖尾1
n += 1;          // 先将p个1清零，然后位p改为1
```

接着，用算术方法执行步骤3，如下：

```
n += 2^(c1 - 1) - 1; // 将拖尾的c1 - 1个0置为1
```

上面的数学运算可缩减为：

```
next = n + (2^c0 - 1) + 1 + (2^(c1 - 1) - 1)
     = n + 2^c0 + 2^(c1 - 1) - 1
```

这种做法的精妙之处在于，只需一两个位操作，代码写起来也很简单。

```
int getNextArith(int n) {
   /* 计算c0和c1，跟之前一样 */
   return n + (1 << c0) + (1 << (c1 - 1)) - 1;
}
```

5. 算术解法：获取前一个数

如果`c1`是拖尾1的个数，`c0`是拖尾1右方全为0的位的个数，则`p = c0 + c1`，前面的`getPrev`可以重新表述如下。

(1) 将位p清零。

(2) 将位p右边的所有位置1。

(3) 将位0到位c0 - 1清零。

上述步骤用算术方法实现如下。为简化起见，这里假定n = 10000011，故c1 = 2且c0 = 5。

```
n -= 2^c1 - 1;       // 清除拖尾1，n变为10000000
n -= 1;              // 翻转拖尾0，n变为01111111
n -= 2^(c0 - 1) - 1; // 翻转最右边(c0-1)个1，n变为01110000
```

由此导出：

```
next   = n - (2^c1 - 1) - 1 - (2^(c0 - 1) - 1)
       = n - 2^c1 - 2^(c0 - 1) + 1
```

和getNextArith一样，实现起来很简单：

```
int getPrevArith(int n) {
   /* 计算c0和c1，跟之前一样 */
   return n - (1 << c1) - (1 << (c0 - 1)) + 1;
}
```

哟！别紧张，在面试中，你用不着写出上面所有解法，至少不会是在没有面试官的大力帮助之下。

5.4　解释代码((n & (n-1)) == 0)的具体含义。(第 56 页)

解法

我们可以由外而内来解决这个问题。

1. (A & B) == 0 是什么意思？

意思是，A和B二进制表示的同一位置绝不会同时为1。因此，如果(n & (n-1)) == 0，则n和n-1就不会有共同的1。

2. 相比 n，n-1 长什么样？

试着动手做一下减法（二进制或十进制），结果会怎么样？

```
   1101011000 [base 2]              593100 [base 10]
 -          1                     -      1
 = 1101010111 [base 2]            = 593099 [base 10]
```

当要将一个数减去1时，需要注意最低有效位。如果最低有效位为1，则变为0，完毕。如果是0，你就必须从高位“借”1。因此，要逐一前往更高的位，将每个位从0改为1，直至找到1为止，并将这个1翻转成0，完毕。

综上，n-1会很像n，只不过n中低位的0在n-1中变为1，n中最低有效位的1在n-1中变为0，示例如下：

```
if     n = abcde1000
then n-1 = abcde0111
```

那么，(n & (n-1)) == 0 究竟表示什么？

n和n-1不存在同一位均为1的情况，因为两者的二进制表示如下：

```
if     n = abcde1000
then n-1 = abcde0111
```

abcde必定全为0，也就是说，n必须像是00001000，因此，n的值是2的某次方。

综上，这个问题的答案为：((n & (n-1)) == 0)检查n是否为2的某次方（或者检查n是否为0）。

5.5　编写一个函数，确定需要改变几个位，才能将整数*A*转成整数*B*。（第 57 页）

解法

这个问题看似复杂，实则非常简单。要解决这个问题，就得设法找出两个数之间有哪些位不同。很简单，使用异或（XOR）操作即可。

在异或操作的结果中，每个1代表*A*和*B*相应位是不一样的。因此，要找出*A*和*B*有多少个不同的位，只要数一数*A*^*B*有几个位为1。

```
int bitSwapRequired(int a, int b) {
   int count = 0;
   for (int c = a ^ b; c != 0; c = c >> 1) {
     count += c & 1;
   }
   return count;
}
```

上面的代码已经很不错了，不过还可以做得更好。上面的做法是不断对c执行移位操作，然后检查最低有效位，但其实可以不断翻转最低有效位，计算要多少次c才会变成0。操作c = c & (c - 1)会清除c的最低有效位。

下面的代码运用了这个方法。

```
public static int bitSwapRequired(int a, int b) {
   int count = 0;
   for (int c = a ^ b; c != 0; c = c & (c-1)) {
     count++;
   }
   return count;
}
```

这段代码是偶尔会在面试中出现的位操作问题。如果之前从未见过，一时很难在面试现场想出来，记住这个技巧，对面试会很有帮助。

5.6　编写程序，交换某个整数的奇数位和偶数位，使用指令越少越好（也就是说，位 0 与位 1 交换，位 2 与位 3 交换，依此类推）。（第 57 页）

解法

跟之前几个问题一样，从不同角度考虑这个问题会很有帮助。要操作一对一对的位，必定困难重重，效率也不见得会高。那么，还有其他什么方式来解决这个问题？

我们可以这么做：先操作奇数位，然后再操作偶数位。有办法将数字n的奇数位左移或右移1位吗？当然有。我们可以用10101010（即0xAA）作为掩码，提取奇数位，并将它们右移1位，移

到偶数位的位置。对于偶数位，可以施以同样的操作。最后，将两次操作的结果合并成一个值。

这种做法共需5条指令，实现代码如下。

```
public int swapOddEvenBits(int x) {
   return ( ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1) );
}
```

上述Java代码实现的是32位整数。如欲处理64位整数，那就需要修改掩码。不过，处理逻辑还是一样的。

5.7　数组A包含 0 到*n*的所有整数，但其中缺了一个。在这个问题中，只用一次操作无法取得数组A里某个整数的完整内容。此外，数组A的元素皆以二进制表示，唯一可用的访问操作是"从A[i]取出第*j*位数据"，该操作的时间复杂度为常数。请编写代码找出那个缺失的整数。你有办法在*O*(*n*)时间内完成吗？（第 57 页）

解法

你可能听到过非常类似的问题：给定一列0到n的数，其中只缺一个数字，把这个数找出来。这个问题解决起来很简单，直接将这列数相加，然后与0到n的总和（即$n * (n+1) / 2$）进行比较。两者差值就是那个缺失的整数。

至于这一题，我们可以根据每个整数的二进制表示，求出它的值，然后计算总和。

这种解法的执行时间为$n * \text{length}(n)$，其中length为n中有多少个位。注意，$\text{length}(n) = \log_2(n)$，因此，真正的执行时间为$O(n \log(n))$，效率不够高！那么，我们该怎么办呢？

其实，我们可以使用类似的解法，不过会更直接第利用每个位的值。

假设有下面这些二进制数（`-----`表示移除的那个数）：

```
00000        00100        01000        01100
00001        00101        01001        01101
00010        00110        01010
-----        00111        01011
```

移除上面那个数会导致最低有效位（记作LSB_1）中1和0的失衡。在0到n的数中，若n为奇数，则0和1的数量相同；若n为偶数，则0比1的数量多一个，也就是说：

```
若n % 2 == 1，则count(0s) = count(1s)
若n % 2 == 0，则count(0s) = 1 + count(1s)
```

由此可见，`count(0s)`必定大于或等于`count(1s)`。

从这列数中移除数值v后，只要检查其他数值的最低有效位，马上就能知道v是偶数还是奇数。

	n % 2 == 0 count(0s) = 1 + count(1s)	n % 2 == 1 count(0s) = count(1s)
v % 2 == 0 LSB_1(v) = 0	a 0 is removed. count(0s) = count(1s)	a 0 is removed. count(0s) < count(1s)
v % 2 == 1 LSB_1(v) = 1	a 1 is removed. count(0s) > count(1s)	a 1 is removed. count(0s) > count(1s)

因此，如果`count(0s) <= count(1s)`，则v为偶数，如果`count(0s) > count(1s)`，则v为奇数。

那么，我们又该如何确定v的下一个位呢？如果这列数中含有v的话，我们就会发现如下规律（其中$count_2$表示第二个最低有效位中0或1的个数）：

$count_2(0s) = count_2(1s)$ 或 $count_2(0s) = 1 + count_2(1s)$

跟前面的例子一样，我们可以推导出v的第二个最低有效位（LSB_2）。

	$count_2(0s) = 1 + count_2(1s)$	$count_2(0s) = count_2(1s)$
$LSB_2(v) == 0$	a 0 is removed. $count_2(0s) = count_2(1s)$	a 0 is removed. $count_2(0s) < count_2(1s)$
$LSB_2(v) == 1$	a 1 is removed. $count_2(0s) > count_2(1s)$	a 1 is removed. $count_2(0s) > count_2(1s)$

同样的，我们可以得出以下结论：

- ❑ 若$count_2(0s) <= count_2(1s)$，则$LSB_2(v) = 0$。
- ❑ 若$count_2(0s) > count_2(1s)$，则$LSB_2(v) = 1$。

重复上述操作可以找出每个位，每次迭代时，我们数出位i中0和1的数量，检查$LSB_i(v)$是0还是1。然后，摒弃$LSB_i(x) != LSB_i(v)$的那些数字。也就是说，若v为偶数，则摒弃奇数，依此类推。

在操作流程的最后，就可得到v所有位的值。在每一次迭代中，我们会查看n个位，然后是$n / 2$个，接着是$n / 4$个，等等。因此，时间复杂度为$O(N)$。

我们还可以更形象地演示整个过程。在第一次迭代时，有下面这些数字：

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

由$count_1(0s) > count_1(1s)$可知$LSB_1(v) = 1$。因此，摒除所有使得$LSB_1(x) != LSB_1(v)$的数$x$。

~~00000~~	~~00100~~	~~01000~~	~~01100~~
00001	00101	01001	01101
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

接着，由$count_2(0s) > count_2(1s)$可知$LSB_2(v) = 1$。因此，摒除所有使得$LSB_2(x) != LSB_2(v)$的数$x$。

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	~~01101~~
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

此时，由$count_3(0s) <= count_3(1s)$可知$LSB_3(v) = 0$。因此，摒除所有使得$LSB_3(x) != LSB_3(v)$的数$x$。

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	~~01101~~
~~00010~~	~~00110~~	~~01010~~	
-----	~~00111~~	01011	

最后只剩下一个数字了，此时$count_4(0s) <= count_4(1s)$，因此$LSB_4(v) = 0$。

摒除所有使得$LSB_4(v)$!= 0的数字之后，我们得到一个空的列表。列表为空之后，可以得到$count_i(0s) <= count_i(1s)$，因此$LSB_i(v) = 0$。换句话说，一旦列表为空，即可将$v$的其余位填为0。

在上面的示例中，整个操作流程将算出`v = 00011`。

下面是该算法的实现代码，我们按位值切分整个数组，借此实现了摒除部分代码。

```
1  public int findMissing(ArrayList<BitInteger> array) {
2     /* bit 0对应于LSB。以此为起点，
3      * 逐步向较高的位推进 */
4     return findMissing(array, 0);
5  }
6
7  public int findMissing(ArrayList<BitInteger> input, int column) {
8     if (column >= BitInteger.INTEGER_SIZE) { // 终止条件与错误条件
9       return 0;
10    }
11    ArrayList<BitInteger> oneBits =
12      new ArrayList<BitInteger>(input.size()/2);
13    ArrayList<BitInteger> zeroBits =
14      new ArrayList<BitInteger>(input.size()/2);
15
16    for (BitInteger t : input) {
17      if (t.fetch(column) == 0) {
18         zeroBits.add(t);
19      } else {
20         oneBits.add(t);
21      }
22    }
23    if (zeroBits.size() <= oneBits.size()) {
24      int v = findMissing(zeroBits, column + 1);
25      return (v << 1) | 0;
26    } else {
27      int v = findMissing(oneBits, column + 1);
28      return (v << 1) | 1;
29    }
30 }
```

在第24和27行，我们会以递归方式计算出v的其他位。然后，再根据$count_1(0s) <= count_1(1s)$是否成立，插入0或1。

5.8　有个单色屏幕存储在一个一维字节数组中，使得 8 个连续像素可以存放在一个字节里。屏幕宽度为w，且w可被 8 整除（即一个字节不会分布在两行上），屏幕高度可由数组长度及屏幕宽度推算得出。请实现一个函数`drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)`，绘制从点(x_1, y)到点(x_2, y)的水平线。（第 57 页）

解法

这个问题有个粗糙的简单解法：用`for`循环迭代，从`x1`到`x2`，一路设定每个像素。但这么做太没劲了，是吧？（况且效率也不高。）

更好的做法是，如果`x1`和`x2`相距甚远，其间包含几个完整字节。只要使用`screen[byte_pos]`

= 0xFF，一次就能设定一整个字节。这条线起点和终点剩余部分的位，可用掩码设定。

```
void drawLine(byte[] screen, int width, int x1, int x2, int y) {
   int start_offset = x1 % 8;
   int first_full_byte = x1 / 8;
   if (start_offset != 0) {
     first_full_byte++;
   }

   int end_offset = x2 % 8;
   int last_full_byte = x2 / 8;
   if (end_offset != 7) {
     last_full_byte--;
   }

   // 设定完整的字节
   for (int b = first_full_byte; b <= last_full_byte; b++) {
     screen[(width / 8) * y + b] = (byte) 0xFF;
   }

   // 创建用于线条起点和终点的掩码
   byte start_mask = (byte) (0xFF >> start_offset);
   byte end_mask = (byte) ~(0xFF >> (end_offset + 1));

   // 设定线条的起点和终点
   if ((x1 / 8) == (x2 / 8)) { // x1和x2位于同一字节
     byte mask = (byte) (start_mask & end_mask);
     screen[(width / 8) * y + (x1 / 8)] |= mask;
   } else {
     if (start_offset != 0) {
        int byte_number = (width / 8) * y + first_full_byte - 1;
        screen[byte_number] |= start_mask;
     }
     if (end_offset != 7) {
        int byte_number = (width / 8) * y + last_full_byte + 1;
        screen[byte_number] |= end_mask;
     }
   }
}
```

务必小心处理这个问题，其中暗藏许多“陷阱”和特殊情况。例如，你必须考虑到x1和x2位于同一字节的情况。只有那些最细心的求职者，才能毫无纰漏地写出这段代码。

9.6 智力题

6.1 有 20 瓶药丸，其中 19 瓶装有 1 克/粒的药丸，余下一瓶装有 1.1 克/粒的药丸。给你一台称重精准的天平，怎么找出比较重的那瓶药丸？天平只能用一次。（第 59 页）

解法

有时候，严格的限制条件有可能反倒是解题的线索。在这个问题中，限制条件是天平只能用一次。

因为天平只能用一次，我们也得以知道一个有趣的事实：一次必须同时称很多药丸，其实更准确地说，是必须从19瓶拿出药丸进行称重。否则，如果跳过两瓶或更多瓶药丸，又该如何区分没称过的那几瓶呢？别忘了，天平只能用一次。

那么，该怎么称重取自多个药瓶的药丸，并确定哪一瓶装有比较重的药丸？假设只有两瓶药丸，其中一瓶的药丸比较重。每瓶取出一粒药丸，称得重量为2.1克，但无从知道这多出来的0.1克来自哪一瓶。我们必须设法区分这些药瓶。

如果从药瓶#1取出一粒药丸，从药瓶#2取出两粒药丸，那么，称得重量为多少呢？结果要看情况而定。如果药瓶#1的药丸较重，则称得重量为3.1克。如果药瓶#2的药丸较重，则称得重量为3.2克。这就是这个问题的解题窍门。

称一堆药丸时，我们会有个"预期"重量。而借由预期重量和实测重量之间的差别，就能得出哪一瓶药丸比较重，前提是从每个药瓶取出不同数量的药丸。

将之前两瓶药丸的解法加以推广，就能得到完整解法：从药瓶#1取出一粒药丸，从药瓶#2取出两粒，从药瓶#3取出三粒，依此类推。如果每粒药丸均重1克，则称得总重量为210克（`1 + 2 + … + 20 = 20 * 21 / 2 = 210`），"多出来的"重量必定来自每粒多0.1克的药丸。

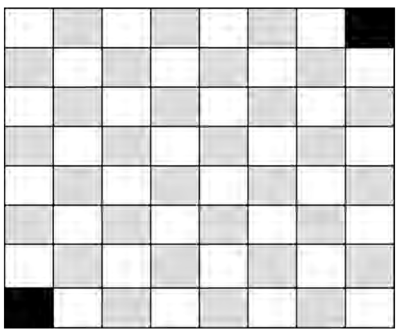
药瓶的编号可由算式`(weight - 210 grams) / 0.1 grams`得出。因此，若这堆药丸称得重量为211.3克，则药瓶#13装有较重的药丸。

6.2　有个 8×8 棋盘，其中对角的角落上，两个方格被切掉了。给定 31 块多米诺骨牌，一块骨牌恰好可以覆盖两个方格。用这 31 块骨牌能否盖住整个棋盘？请证明你的答案（提供范例，或证明为什么不可能）。（第 59 页）

解法

乍一看，似乎是可以盖住的。棋盘大小为8×8，共有64个方格，但其中两个方格已被切掉，因此只剩62个方格。31块骨牌应该刚好能盖住整个棋盘，对吧？

尝试用骨牌盖住第1行，而第1行只有7个方格，因此有一块骨牌必须铺至第2行。而用骨牌盖住第2行时，我们又必须将一块骨牌铺至第3行。



要盖住每一行，总有一块骨牌必须铺至下一行。无论尝试多少次、多少种方法，我们都无法成功铺下所有骨牌。

其实，还有更简洁更严谨的证明说明为什么不可能。棋盘原本有32个黑格和32个白格。将对角角落上的两个方格（相同颜色）切掉，棋盘只剩下30个同色的方格和32个另一种颜色的方格。为方便论证起见，我们假定棋盘上剩下30个黑格和32个白格。

放在棋盘上的每块骨牌必定会盖住一个白格和一个黑格。因此，31块骨牌正好盖住31个白格和31个黑格。然而，这个棋盘只有30个黑格和32个白格，所以，31块骨牌盖不住整个棋盘。

6.3 有两个水壶，容量分别为 5 夸脱（美制：1 夸脱=0.946 升，英制：1 夸脱=1.136 升）和 3 夸脱，若水的供应不限量（但没有量杯），怎么用这两个水壶得到刚好 4 夸脱的水？注意，这两个水壶呈不规则形状，无法精准地装满“半壶”水。（第 59 页）

解法

根据题意，我们只能使用这两个水壶，不妨随意把玩一番，把水倒来倒去，可以得到如下顺序组合：

5夸脱	3夸脱	注　　解
5	0	装满5夸脱水壶
2	3	用5夸脱水壶里的水装满3夸脱水壶
0	2	把5夸脱水壶里的水倒入3夸脱水壶
5	2	装满5夸脱水壶
4	3	用5夸脱水壶里的水填满3夸脱水壶
4		搞定！准确量得4夸脱。

注意，许多智力题其实都隐含数学或计算机科学的背景，这个问题也不例外。只要这两个水壶的容量互质（即两个数没有共同的质因子），我们就能找出一种倒水的顺序组合，量出1到2个水壶容量总和（含）之间的任意水量。

6.4 有个岛上住着一群人，有一天来了个游客，定了一条奇怪的规矩：所有蓝眼睛的人都必须尽快离开这个岛。每晚 8 点会有一个航班离岛。每个人都看得见别人眼睛的颜色，但不知道自己的（别人也不可以告知）。此外，他们不知道岛上到底有多少人是蓝眼睛的，只知道至少有一个人的眼睛是蓝色的。所有蓝眼睛的人要花几天才能离开这个岛？（第 59 页）

解法

下面将采用简单构造法。假定这个岛上一共有n人，其中c人有蓝眼睛。由题目可知，$c > 0$。

1. 情况 $c = 1$：只有一人是蓝眼睛的

假设岛上所有人都是聪明的，蓝眼睛的人四处观察之后，发现没有人是蓝眼睛的。但他知道至少有一人是蓝眼睛的，于是就能推导出自己一定是蓝眼睛的。因此，他会搭乘当晚的飞机离开。

2. 情况 $c = 2$：只有两人是蓝眼睛的

两个蓝眼睛的人看到对方，并不确定c是1还是2，但是由上一种情况，他们知道，如果$c = 1$，

那个蓝眼睛的人第一晚就会离岛。因此，发现另一个蓝眼睛的人仍在岛上，他一定能推断出$c=2$，也就意味着他自己也是蓝眼睛的。于是，两个蓝眼睛的人都会在第二晚离岛。

3. 情况 $c>2$：一般情况

逐步提高c时，我们可以看出上述逻辑仍旧适用。如果$c=3$，那么，这三个人会立即意识到有2到3人是蓝眼睛的。如果有两人是蓝眼睛的，那么这两人会在第二晚离岛。因此，如果过了第二晚另外两人还在岛上，每个蓝眼睛的人都能推断出$c=3$，因此这三人都有蓝眼睛。他们会在第三晚离岛。

不论c为什么值，都可以套用这个模式。所以，如果有c人是蓝眼睛的，则所有蓝眼睛的人要用c晚才能离岛，且都在同一晚离开。

6.5　有栋建筑物高 100 层。若从第*N*层或更高的楼层扔下来，鸡蛋就会破掉。若从第*N*层以下的楼层扔下来则不会破掉。给你 2 个鸡蛋，请找出*N*，并要求最差情况下扔鸡蛋的次数为最少。（第 59 页）

解法

我们发现，无论怎么扔鸡蛋1（`Egg 1`），鸡蛋2（`Egg 2`）都必须在“破掉那一层”和下一个不会破掉的最高楼层之间，逐层扔下楼（从最低的到最高的）。例如，若鸡蛋1从5层和10层楼扔下没破掉，但从15层扔下时破掉了，那么，在最差情况下，鸡蛋2必须尝试从11、12、13和14层扔下楼。

具体做法

首先，让我们试着从10层开始扔鸡蛋，然后是20层，等等。

- 如果鸡蛋1第一次扔下楼（10层）就破掉了，那么，最多需要扔10次。
- 如果鸡蛋1最后一次扔下楼（100层）才破掉，那么，最多要扔19次（10、20、…、90、100层，然后是91到99层）。

这么做也挺不错，但我们只考虑了绝对最差情况。我们应该进行“负载均衡”，让这两种情况下扔鸡蛋的次数更均匀。

我们的目标是设计一种扔鸡蛋的方法，使得扔鸡蛋1时，不论是在第一次还是最后一次扔下楼才破掉，次数越稳定越好。

(1) 完美负载均衡的方法应该是，扔鸡蛋1的次数加上扔鸡蛋2的次数，不论什么时候都一样，不管鸡蛋1是从哪层楼扔下时破掉的。

(2) 若有这种扔法，每次鸡蛋1多扔一次，鸡蛋2就可以少扔一次。

(3) 因此，每丢一次鸡蛋1，就应该减少鸡蛋2可能需要扔下楼的次数。例如，如果鸡蛋1先从20层往下扔，然后从30层扔下楼，此时鸡蛋2可能就要扔9次。若鸡蛋1再扔一次，我们必须让鸡蛋2扔下楼的次数降为8次。也就是说，我们必须让鸡蛋1从39层扔下楼。

(4) 由此可知，鸡蛋1必须从X层开始往下扔，然后再往上增加$X-1$层……直至到达100层。

(5) 求解方程式$X+(X-1)+(X-2)+\cdots+1=100$，得到$X(X+1)/2=100 \rightarrow X=14$。

我们先从14层开始，然后是27层，接着是39层，依此类推，最差情况下鸡蛋要扔14次。

正如解决其他许多最大化/最小化的问题一样，这类问题的关键在于“平衡最差情况”。

6.6　走廊上有 100 个关上的储物柜。有个人先是将 100 个柜子全都打开。接着，每数两个柜子关上一个。然后，在第三轮时，再每隔两个就切换第三个柜子的开关状态（也就是将关上的柜子打开，将打开的关上）。照此规律反复操作 100 次，在第*i*轮，这个人会每数*i*个就切换第*i*个柜子的状态。当第 100 轮经过走廊时，只切换第 100 个柜子的开关状态，此时有几个柜子是开着的？（第 59 页）

解法

要解决这个问题，我们必须弄清楚所谓切换储物柜开关状态是什么意思。这有助于我们推断最终哪些柜子是开着的。

1. 问题：柜子会在哪几轮切换状态（开或关）？

柜子*n*会在*n*的每个因子（包括1和*n*本身）对应的那一轮切换状态。也就是说，柜子15会在第1、3、5和15轮开或关一次。

2. 问题：柜子什么时候还是开着的？

如果因子个数（记作*x*）为奇数，则这个柜子是开着的。你可以把一对因子比作开和关，若还剩一个因子，则柜子就是开着的。

3. 问题：*x* 什么时候为奇数？

若*n*为完全平方数，则*x*的值为奇数。理由如下：将n的两个互补因子配对。例如，如*n*为36，则因子配对情况为：(1, 36)、(2, 18)、(3, 12)、(4, 9)、(6, 6)。注意，(6, 6)其实只有一个因子，因此*n*的因子个数为奇数。

4. 问题：有多少个完全平方数？

一共有10个完全平方数，你可以数一数（1、4、9、16、25、36、49、64、81、100），或者，直接列出1到10的平方：

```
1*1, 2*2, 3*3, ..., 10*10
```

因此，最后共有10个柜子是开着的。

9.7　数学与概率

7.1　有个篮球框，下面两种玩法可任选一种。

玩法 1：一次出手机会，投篮命中得分。

玩法 2：三次出手机会，必须投中两次。

如果p是某次投篮命中的概率，则p的值为多少时，才会选择玩法 1 或玩法 2？（第 63 页）

解法

要解此题，我们可以直接运用概率论，比较赢得各种玩法的概率。

1. 赢得玩法 1 的概率：

根据定义，赢得玩法1的概率为p。

2. 赢得玩法 2 的概率：

令$s(k,n)$为n次投篮准确投中k次的概率，赢得玩法2的概率是三投两中或三投三中的概率。换句话说：

P(获胜) = $s(2,3) + s(3,3)$

三投三中的概率为：

$s(3,3) = p^3$

三投两中的概率为：

P(第1、2次投中，第3次未投中)
+ P(第1、3次投中，第2次未投中)
+ P(第1次未投中，第2、3次投中)

$= p * p * (1-p) + p * (1-p) * p + (1-p) * p * p$

$= 3\ (1-p)\ p^2$

两者概率相加，可以得到：

$= p^3 + 3\ (1-p)\ p^2$

$= p^3 + 3p^2-3p^3$

$= 3p^2-2p^3$

3. 该选择哪种玩法？

若P(玩法1) > P(玩法2)，则该选择玩法1：

$p > 3p^2-2p^3$

$1 > 3p-2p^2$

$2p^2-3p + 1 > 0$

$(2p-1)(p-1) > 0$

左边两项必须同为正数或同为负数。显然，$p < 1$，故$p-1 < 0$，也即这两项必须同为负数。

$2p-1 < 0$

$2p < 1$

$p < .5$

综上，若p <0.5，则应该选择玩法1。若p = 0、0.5或1，则P(玩法1) = P(玩法2)，选哪种玩法都一样，因为赢得两种玩法的概率相等。

7.2　三角形的三个顶点上各有一只蚂蚁。如果蚂蚁开始沿着三角形的边爬行，两只或三只蚂蚁撞在一起的概率有多大？假定每只蚂蚁会随机选一个方向，每个方向被选到的几率相等，而且三只蚂蚁的爬行速度相同。

类似问题：在n个顶点的多边形上有n只蚂蚁，求出这些蚂蚁发生碰撞的概率。（第 63 页）

解法

当其中两只蚂蚁互相朝着对方而行，就会发生碰撞。因此，蚂蚁不发生碰撞的前提是，它们都朝着同一方向爬行（顺时针或逆时针）。我们可以算出这种情况的概率，然后再反推出问题的答案。

每只蚂蚁可以朝两个方向爬行，一共有3只蚂蚁，它们不发生碰撞的概率位：

$P(\text{顺时针}) = (\frac{1}{2})^3$

$P(\text{逆时针}) = (\frac{1}{2})^3$

$P(\text{同方向}) = (\frac{1}{2})^3 + (\frac{1}{2})^3 = \frac{1}{4}$

因此，发生碰撞的概率就是蚂蚁不朝着同方向爬行的概率：

$P(\text{碰撞}) = 1 - P(\text{同方向}) = 1 - (\frac{1}{4}) = \frac{3}{4}$

若要将这个方法推广至n个顶点的多边形，同样的，蚂蚁也只有以顺时针或逆时针同方向爬行才不致相撞，但总共有2^n种爬行方式。综上，发生碰撞的概率为：

$P(\text{顺时针}) = (\frac{1}{2})^n$

$P(\text{逆时针}) = (\frac{1}{2})^n$

$P(\text{同方向}) = 2(\frac{1}{2})^n = (\frac{1}{2})^{n-1}$

$P(\text{碰撞}) = 1 - P(\text{同方向}) = 1 - (\frac{1}{2})^{n-1}$

7.3 给定直角坐标系上的两条线，确定这两条线会不会相交。（第 63 页）

解法

此题有很多不确定的地方：两条线的格式是什么？两条线实为同一条怎么处理？这些含糊不清的地方最好跟面试官讨论一下。

下面将做出以下假设：

- 若两条线是相同的（斜率和y轴截距相等），则认为这两条线相交；
- 我们可以决定线的数据结构。

两条线若不平行则必相交。因此，要检查两条线相交与否，我们只需检查两者的斜率是否相同，或是否为同一条。

实现代码如下：

```
public class Line {
   static double epsilon = 0.000001;
   public double slope;
   public double yintercept;

   public Line(double s, double y) {
     slope = s;
     yintercept = y;
   }

   public boolean intersect(Line line2) {
     return Math.abs(slope - line2.slope) > epsilon ||
            Math.abs(yintercept - line2.yintercept) < epsilon;
```

```
    }
}
```

遇到这类问题时，务请注意以下几点。

- 多提问。此题存在诸多不明之处，多提问以厘清问题。许多面试官会故意提些模糊的问题，考察你是否会说明自己的假设条件。
- 尽量设计并使用数据结构，借此展示你了解并注重面向对象设计。
- 仔细考虑要怎么设计数据结构来表示一条线。选择多多，各有优劣，必须权衡取舍。选择一种数据结构，并说明理由。
- 不要假设斜率和y轴截距就是整数。
- 了解浮点表示法的限制。切记不要用==检查浮点数是否相等，而是应该检查两者差值是否小于某个极小值（如上面代码中的epsilon值）。

7.4　编写方法，实现整数的乘法、减法和除法运算。只允许使用加号。（第 63 页）

解法

此题只允许使用加号运算符。对于每个子问题，最好深入思考这些运算的本质，或者如何用其他运算表示（加法或已实现的运算）。

1. 减法

怎样才能用加法表示减法？这个问题看起来直截了当，运算$a - b$跟$a + (-1) * b$是一回事。不过，根据题意，不得使用乘号（*），因此我们必须实现一个取反（negate）的函数。

```
/* 正号变负号，负号变正号 */
public static int negate(int a) {
   int neg = 0;
   int d = a < 0 ? 1 : -1;
   while (a != 0) {
     neg += d;
     a += d;
   }
   return neg;
}

/* 两数相减相当于对b取反，然后将两数相加 */
public static int minus(int a, int b) {
   return a + negate(b);
}
```

要对数值k的取反，只需将-1连加k次。

2. 乘法

加法和乘法之间关系也同样一目了然，a乘以b其实就是将a连加b次。

```
/* 将a连加b次，实现a乘b */
public static int multiply(int a, int b) {
   if (a < b) {
```

```
     return multiply(b, a); // 若b < a，算法会比较快
   }
   int sum = 0;
   for (int i = abs(b); i > 0; i--) {
     sum += a;
   }
   if (b < 0) {
     sum = negate(sum);
   }
   return sum;
}

/* 返回绝对值 */
public static int abs(int a) {
   if (a < 0) {
     return negate(a);
   } else {
     return a;
   }
}
```

在上面的代码中，有个地方必须妥善处理，就是负数的乘法。若b为负数，则需将`sum`的值正负反一下。因此，这段代码实际上是这么回事：

```
multiply(a, b) <-- abs(b) * a * (-1 if b < 0).
```

我们还实现了一个简单的`abs`辅助函数。

3. 除法

在减、乘、除三种运算中，除法无疑是最难的。好在我们可以利用已有的`multiply`、`subtract`和`negate`等方法实现`divide`。

除法要做的是计算$x = a / b$中的x。或者，换个角度来说，找到x，使得$a = bx$。这么一来，经过变换，这个问题就可以用之前已实现的乘法运算实现。

我们可以将b不断乘以逐级变大的值，直至得到a。这么做非常低效，特别是前面的`multiply`实现含有大量加法运算。

或者，我们可以好好利用等式$a = xb$，将b与它自身连加直至得到a，就能算出x。b与自身连加的次数就等于x的值。

当然，a不一定能被b整除，这也没关系。这个问题要求实现的是整数除法，本来就应该对结果向下舍入（`floor`）。

下面是这个算法的实现代码。

```
public int divide(int a, int b)
     throws java.lang.ArithmeticException {
   if (b == 0) {
     throw new java.lang.ArithmeticException("ERROR");
   }
   int absa = abs(a);
   int absb = abs(b);

```

```
    int product = 0;
    int x = 0;
    while (product + absb <= absa) { /* 不要超过a */
      product += absb;
      x++;
    }

    if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
      return x;
    } else {
      return negate(x);
    }
}
```

解决此题时，应当注意以下几点。

- 多想想乘法和除法的本质，以逻辑思考的方式解题，这么做很管用。记住，所有（好的）面试题都能以逻辑、系统的方式解出来！
- 面试官想找的就是这种能以逻辑思考逐步解决问题的人。
- 这是个让你展示自己能够写出干净代码的绝佳问题，特别是显示出你复用代码的能力。例如，如果写代码时没有把negate独立出来，但要是发现相关代码使用多次，就应该将这些代码写成一个方法。
- 写代码时要小心假设。不要假设所有数都是正数，也不该假设*a*会比*b*大。

7.5　在二维平面上，有两个正方形，请找出一条直线，能够将这两个正方形对半分。假定正方形的上下两条边与*x*轴平行。（第 63 页）

解法

在着手解题之前，有必要思考一下题中一条“线”的准确含义。一条线是由斜率和*y*轴截距确定？还是由这条线上的任意两点定义？抑或，所谓的线其实是线段，以正方形的边作为起点和终点？

其中第三种情况会让此题变得更有意思一些，因此这里假设：这条线的端点应该落在正方形的边上。在面试中，你应该与面试官讨论假设条件。

要将两个正方形对半分，这条线必须连接两个正方形的中心点。利用 $\text{slope} = \frac{(y_1 - y_2)}{(x_1 - x_2)}$ 就能算出斜率，以两个中心点算出斜率后，就能以同一公式求得线段的起点和终点。

在下面的代码中，假设原点(0, 0)位于左上角。

```
public class Square {
    ...
    public Point middle() {
      return new Point((this.left + this.right) / 2.0,
                       (this.top + this.bottom) / 2.0);
    }

    /* 返回连接mid1和mid2的线段与square 1
     * 的边相交的点，也就是说，从mid2到mid1
```

```
 * 画一条线，一直延伸直至碰到square 1的
 * 边
 */
public Point extend(Point mid1, Point mid2, double size) {
  /* 找出线段mid2 -> mid1的方向 */
  double xdir = mid1.x < mid2.x ? -1 : 1;
  double ydir = mid1.y < mid2.y ? -1 : 1;

  /* 若mid1和mid2的x坐标相同，计算斜率时
   * 会抛出除零异常，因此这里要做特别的处理
   */
  if (mid1.x == mid2.x) {
     return new Point(mid1.x, mid1.y + ydir * size / 2.0);
  }

  double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
  double x1 = 0;
  double y1 = 0;

  /* 利用算式(y1 - y2) / (x1 - x2)计算斜率（slope）。
   * 注意，若斜率很“陡峭”（>1），那么线段的终点将会
   * 碰到y轴上距离中心点size / 2的位置。若斜率
   * 不陡峭（<1），那么线段的终点将碰到x轴上距
   * 离中心点size / 2的位置
   */
  if (Math.abs(slope) == 1) {
     x1 = mid1.x + xdir * size / 2.0;
     y1 = mid1.y + ydir * size / 2.0;
  } else if (Math.abs(slope) < 1) {
     x1 = mid1.x + xdir * size / 2.0;
     y1 = slope * (x1 - mid1.x) + mid1.y;
  } else {
     y1 = mid1.y + ydir * size / 2.0;
     x1 = (y1 - mid1.y) / slope + mid1.x;
  }
  return new Point(x1, y1);
}

public Line cut(Square other) {
  /* 计算两个中心点之间的线段与正方形的边相交的位置 */
  Point point_1 = extend(this.middle(), other.middle(), this.size);
  Point point_2 = extend(this.middle(), other.middle(), -1 * this.size);
  Point point_3 = extend(other.middle(), this.middle(), other.size);
  Point point_4 = extend(other.middle(), this.middle(), -1 * other.size);

  /* 在上面这些点中，找出线段的起点和终点。起点以最左边且在上方的为准，
  * 终点以最右边且在下方的为准 */
  Point start = point_1;
  Point end = point_1;
  Point[] points = {point_2, point_3, point_4};
  for (int i = 0; i < points.length; i++) {
     if (points[i].x < start.x || (points[i].x == start.x && points[i].y < start.y)) {
        start = points[i];
    } else if (points[i].x > end.x || (points[i].x == end.x && points[i].y > end.y)) {
```

```
          end = points[i];
        }
      }

      return new Line(start, end);
    }
}
```

此题意在考察你写代码有多细心，毕竟写代码时很容易漏掉一些特殊情况，比如两个正方形的中心点重合。着手解题之前，我们就应该列出这些特殊情况，并确保予以妥善处理。解题时，测试必须仔细、全面。

7.6　在二维平面上，有一些点，请找出经过点数最多的那条线。（第 63 页）

解法

此题乍一看很简单，老实说，确实有点。

我们只需在任意两点之间“画”一条无限长的直线（也即不是线段），并利用散列表追踪哪条直线出现次数最多。这种做法的时间复杂度为$O(N^2)$，因为一共有N^2条线段。

我们将用斜率和y轴截距而不是两个点来表示一条线，这样一来，检查(x_1, y_1)、(x_2, y_2)确定的直线是否等于(x_3, y_3)到(x_4, y_4)的直线就相对简单。

要找到出现次数最多的直线，只需迭代遍历所有线段，并用散列表数出每条直线出现的次数。够简单吧！

不过，其中有个地方比较棘手。首先，我们定义，若两条直线的斜率和y轴截距相同，则这两条直线相等。接着，我们会基于这些值（确切地说，是基于斜率）对直线进行散列。问题是浮点数不一定能用二进制精确表示。对此，我们的解决办法是检查两个浮点数的差值是否在某个极小值（epsilon）内。

对散列表而言，这又意味着什么呢？这意味着，斜率“相等”的两条直线，散列值未必相同。为此，我们将把斜率减去一个极小值，并以得到的结果flooredSlope作为散列键。然后，要取得所有可能相等的直线，我们会搜索三个位置：flooredSlope、flooredSlope - epsilon和flooredSlope + epsilon。这能确保我们已检查了所有可能相等的直线。

```
Line findBestLine(GraphPoint[] points) {
  Line bestLine = null;
  int bestCount = 0;
  HashMap<Double, ArrayList<Line>> linesBySlope =
          new HashMap<Double, ArrayList<Line>>();

  for (int i = 0; i < points.length; i++) {
    for (int j = i + 1; j < points.length; j++) {
      Line line = new Line(points[i], points[j]);
      insertLine(linesBySlope, line);
      int count = countEquivalentLines(linesBySlope, line);
      if (count > bestCount) {
        bestLine = line;
        bestCount = count;
      }
```

```java
    }
  }
  return bestLine;
}

int countEquivalentLines(ArrayList<Line> lines, Line line) {
  if (lines == null) return 0;
  int count = 0;
  for (Line parallelLine : lines) {
    if (parallelLine.isEquivalent(line) count++;
  }
  return count;
}

int countEquivLines(HashMap<Double, ArrayList<Line>> linesBySlope, Line line) {
  double key = Line.floorToNearestEpsilon(line.slope);
  double eps = Line.epsilon;
  int count = countEquivalentLines(linesBySlope.get(key), line) +
              countEquivalentLines(linesBySlope.get(key - eps), line) +
              countEquivalentLines(linesBySlope.get(key + eps), line);
  return count;
}

void insertLine(HashMap<Double, ArrayList<Line>> linesBySlope,
  Line line) {
  ArrayList<Line> lines = null;
  double key = Line.floorToNearestEpsilon(line.slope);
  if (!linesBySlope.containsKey(key)) {
  lines = new ArrayList<Line>();
  linesBySlope.put(key, lines);
  } else {
    lines = linesBySlope.get(key);
  }
  lines.add(line);
}

public class Line {
  public static double epsilon = .0001;
  public double slope, intercept;
  private boolean infinite_slope = false;

  public Line(GraphPoint p, GraphPoint q) {
    if (Math.abs(p.x - q.x) > epsilon) { // 若两个点的x坐标不同
      slope = (p.y - q.y) / (p.x - q.x); // 计算斜率
      intercept = p.y - slope * p.x; // 利用y=mx+b计算y轴截距
    } else {
      infinite_slope = true;
      intercept = p.x; // x轴截距，因为斜率无穷大
    }
  }

  public static double floorToNearestEpsilon(double d) {
    int r = (int) (d / epsilon);
    return ((double) r) * epsilon;
```

```
  }

  public boolean isEquivalent(double a, double b) {
    return (Math.abs(a - b) < epsilon);
  }

  public boolean isEquivalent(Object o) {
    Line l = (Line) o;
    if (isEquivalent(l.slope, slope) &&
        isEquivalent(l.intercept, intercept) &&
        (infinite_slope == l.infinite_slope)) {
      return true;
    }
    return false;
  }
}
```

计算直线的斜率务必小心谨慎。直线有可能完全垂直，也即它没有y轴截距且斜率无穷大。我们可以用单独的标记（`infinite_slope`）跟踪记录。在`equals`方法中，必须检查这个条件。

7.7　有些数的素因子只有 3、5、7，请设计一个算法，找出其中第k个数。（第 63 页）

解法

根据定义，这些数字看起来都像是$3^a * 5^b * 7^c$。

下面先列出符合该形式的数字，此题要求找出这种数字里的第k个。

1	-	$3^0 * 5^0 * 7^0$
3	3	$3^1 * 5^0 * 7^0$
5	5	$3^0 * 5^1 * 7^0$
7	7	$3^0 * 5^0 * 7^1$
9	3*3	$3^2 * 5^0 * 7^0$
15	3*5	$3^1 * 5^1 * 7^0$
21	3*7	$3^1 * 5^0 * 7^1$
25	5*5	$3^0 * 5^2 * 7^0$
1	-	$3^0 * 5^0 * 7^0$
27	3*9	$3^3 * 5^0 * 7^0$
35	5*7	$3^0 * 5^1 * 7^1$
45	5*9	$3^2 * 5^1 * 7^0$
49	7*7	$3^0 * 5^0 * 7^2$
63	3*21	$3^2 * 5^0 * 7^1$

由于$3^{a-1} * 5^b * 7^c < 3^a * 5^b * 7^c$，因此$3^{a-1} * 5^b * 7^c$必定已在我们的列表中出现过。实际上，下面这些值已在列表中出现过了：

- ❑ $3^{a-1} * 5^b * 7^c$
- ❑ $3^a * 5^{b-1} * 7^c$

- $3^a * 5^b * 7^{c-1}$

另一种思路是，所有数字都可以表示成如下形式：

- 3 * (列表中之前出现的某个数)
- 5 * (列表中之前出现的某个数)
- 7 * (列表中之前出现的某个数)

由此可知，A_k可以表示为(3、5或7) * ($\{A_1, ..., A_{k-1}\}$中的某个值)。另外，根据定义可知，A_k是列表中的下一个数。因此，A_k将是最小的新增数字（其余更小的数已在$\{A_1, ..., A_{k-1}\}$中），可以通过将列表中的每个值与3、5或7相乘得到。

怎样才能找到A_k？实际上，我们可以将列表中的数字与3、5和7相乘，找出还未加入列表的最小数。这种解法的时间复杂度为$O(k^2)$。不算太糟，不过我想还可以做得更好。

之前我们曾试着从列表中的元素“拉出”A_k（将这些元素与3、5和7相乘），其实可以换个思路，可以让列表中的元素“推出”三个后续值，也就是说，列表中的每个数A_i终将以下列形式出现：

- $3 * A_i$
- $5 * A_i$
- $7 * A_i$

照着这个思路事先做好准备，每次要将A_i加入列表时，就用某个临时列表存放$3A_i$、$5A_i$和$7A_i$三个值。要产生A_{i+1}时，我们会搜索这个临时列表，找出最小的值。

我们的代码大致如下：

```java
public static int removeMin(Queue<Integer> q) {
   int min = q.peek();
   for (Integer v : q) {
     if (min > v) {
        min = v;
     }
   }
   while (q.contains(min)) {
     q.remove(min);
   }
   return min;
}

public static void addProducts(Queue<Integer> q, int v) {
   q.add(v * 3);
   q.add(v * 5);
   q.add(v * 7);
}

public static int getKthMagicNumber(int k) {
   if (k < 0) return 0;

   int val = 1;
   Queue<Integer> q = new LinkedList<Integer>();
   addProducts(q, 1);
   for (int i = 0; i < k; i++) {
```

```
      val = removeMin(q);
      addProducts(q, val);
    }
    return val;
  }
```

相比第一种解法，这个算法确实要好得多，但仍不够完美。

为了产生新元素A_i，我们会搜索一整个链表，其中每个元素类似如下形式之一：

- 3 * 之前的元素
- 5 * 之前的元素
- 7 * 之前的元素

我们还可以优化掉哪些无谓的操作？

假设有如下列表。

$$q_6 = \{7A_1, 5A_2, 7A_2, 7A_3, 3A_4, 5A_4, 7A_4, 5A_5, 7A_5\}$$

要在列表中查找最小值时，先检查$7A_1 < min$是否成立，然后检查$7A_5 < min$。这看起来有点笨拙，是不是？既然已经知道$A_1 < A_5$，因此只需检查$7A_1$即可。

若从一开始就按常数因子将列表分组存放，那就只需检查3、5和7倍数的第一个，后续元素一定比第一个元素大。

也就是说，上面的列表应该如下：

$$Q36 = \{3A_4\}$$

$$Q56 = \{5A_2, 5A_4, 5A_5\}$$

$$Q76 = \{7A_1, 7A_2, 7A_3, 7A_4, 7A_5\}$$

要求得最小值，我们只需检查各个队列的队首元素。

$$y = \min(Q3.head(), Q5.head(), Q7.head())$$

求出y后，就要把$3y$插入Q3、$5y$插入Q5、$7y$插入Q7。不过，只有这些元素在其他列表中不存在时，我们才会将它们插入列表。

举个例子，为什么$3y$可能已经存在某个队列中？很简单，如果y是从Q7拉出来的，就表示$y = 7x$，x是某个较小的值。如果$7x$是最小值，那么，我们一定碰到过$3x$。碰到$3x$时会怎么做呢？我们会将$7 * 3x$插入Q7。注意，$7 * 3x = 3 * 7x = 3y$。

换句话说，如果从Q7拉出一个元素，它看起来像7 * suffix，而我们知道已处理过3 * suffix和5 * suffix。处理3 * suffix时，将7 * 3 * suffix插入Q7。而处理5 * suffix时，我们知道已经将7 * 5 * suffix插入Q7。至此，唯一还未碰到的值是7 * 7 * suffix，因此我们只会将7 * 7 * suffix插入Q7。

下面我们会举例说明，真正做到心知肚明。

```
一开始：
        Q3 = 3
        Q5 = 5
        Q7 = 7
取出min = 3，Q3插入3*3，Q5插入5*3，Q7插入7*3。
        Q3 = 3*3
        Q5 = 5, 5*3
```

```
        Q7 = 7, 7*3
取出min = 5，3*5重复了，因为我们已经处理过5*3。Q5插入5*5，Q7插入7*5。
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7, 7*3, 7*5.
取出min = 7，3*7和5*7重复了，因为已处理过7*3和7*5。Q7插入7*7。
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7*3, 7*5, 7*7
取出min = 3*3 = 9，Q3插入3*3*3，Q5插入3*3*5，Q7插入3*3*7。
        Q3 = 3*3*3
        Q5 = 5*3, 5*5, 5*3*3
        Q7 = 7*3, 7*5, 7*7, 7*3*3
取出min = 5*3 = 15，3*(5*3)重复了，因为已处理过5*(3*3)。Q5插入5*5*3，Q7插入7*5*3。
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
        Q7 = 7*3, 7*5, 7*7, 7*3*3, 7*5*3
取出min = 7*3 = 21，3*(7*3)和5*(7*3)重复了，因为已处理过7*(3*3)和7*(5*3)。Q7插入7*7*3。
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
        Q7 = 7*5, 7*7, 7*3*3, 7*5*3, 7*7*3
```

此题解法的伪码如下。

(1) 初始化array和队列：Q3、Q5和Q7。

(2) 将1插入array。

(3) 分别将1*3、1*5和1*7插入Q3、Q5和Q7。

(4) 令*x*为Q3、Q5和Q7中的最小值。将*x*添加至array尾部。

(5) 若x存在于：

- Q3，则将*x**3、*x**5和*x**7放入Q3、Q5和Q7，从Q3移除*x*。
- Q5，则将*x**5和*x**7放入Q5和Q7，从Q5移除*x*。
- Q7，则只将*x**7放入Q7，从Q7移除*x*。

(6) 重复步骤4~6，直至找到第*k*个元素。

下面是该算法的实现代码。

```
public static int getKthMagicNumber(int k) {
   if (k < 0) {
     return 0;
   }
   int val = 0;
   Queue<Integer> queue3 = new LinkedList<Integer>();
   Queue<Integer> queue5 = new LinkedList<Integer>();
   Queue<Integer> queue7 = new LinkedList<Integer>();
   queue3.add(1);

   /* 从0到k的迭代 */
   for (int i = 0; i <= k; i++) {
     int v3 = queue3.size() > 0 ? queue3.peek() :
                                  Integer.MAX_VALUE;
     int v5 = queue5.size() > 0 ? queue5.peek() :
```

```
                                  Integer.MAX_VALUE;
      int v7 = queue7.size() > 0 ? queue7.peek() :
                                  Integer.MAX_VALUE;
      val = Math.min(v3, Math.min(v5, v7));
      if (val == v3) { // 放入队列3、队列5和队列7
         queue3.remove();
         queue3.add(3 * val);
         queue5.add(5 * val);
      } else if (val == v5) { // 放入队列5和队列7
         queue5.remove();
         queue5.add(5 * val);
      } else if (val == v7) { // 放入队列7
         queue7.remove();
      }
      queue7.add(7 * val); // 总是放入队列7
   }
   return val;
}
```

碰到这个问题时，尽最大努力去解决，虽然问题确实有难度。你可以先从蛮力法开始（有挑战性，但不那么棘手），然后试着不断优化。或者，试着从这些数中找出规律。

当你解题卡壳时，面试官有可能会帮你一把。不管怎样，绝不要放弃！大声说出你的思路、疑问，并解释你的思考过程。面试官或许就会介入指导一下。

记住，面试官并不期待你给出完美无缺的解法，而是会对照其他求职者来评估你的表现。面对刁钻的问题，大家都需拼尽全力。

9.8　面向对象设计

8.1　请设计用于通用扑克牌的数据结构。并说明你会如何创建该数据结构的子类，实现“二十一点”游戏。（第 66 页）

解法

首先，看得出来所谓的“通用”扑克牌隐含有不少信息。这里的“通用”可以指能用来玩扑克牌游戏的标准扑克牌组，也可以扩展为Uno牌或棒球卡。面试时记得询问面试官“通用”的具体含义，这点很重要。

假设面试官说清楚了，这是一副标准纸牌，一共52张，就如同你在二十一点或扑克牌游戏中使用的牌组。这样一来，整个设计大致如下：

```
public enum Suit {
   Club (0), Diamond (1), Heart (2),Spade (3);
   private int value;
   private Suit(int v) { value = v; }
   public int getValue() { return value; }
   public static Suit getSuitFromValue(int value) { ... }
}

```

```java
public class Deck <T extends Card> {
   private ArrayList<T> cards; // 所有牌，包括已经发出去的，还未发出去的
   private int dealtIndex = 0; // 标示第一张还未发出去的牌

   public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }

   public void shuffle() { ... }
   public int remainingCards() {
     return cards.size() - dealtIndex;
   }
   public T[] dealHand(int number) { ... }
   public T dealCard() { ... }
}

public abstract class Card {
   private boolean available = true;

   /* 牌面的数字或人头，数字2到10，11为杰克，
    * 12为皇后，13位国王，1为Ace */
   protected int faceValue;
   protected Suit suit;

   public Card(int c, Suit s) {
     faceValue = c;
     suit = s;
   }

   public abstract int value();

   public Suit suit() { return suit; }

   /* 检查这张牌是否发给某个人 */
   public boolean isAvailable() { return available; }
   public void markUnavailable() { available = false; }

   public void markAvailable() { available = true; }
}

public class Hand <T extends Card> {
   protected ArrayList<T> cards = new ArrayList<T>();

   public int score() {
     int score = 0;
     for (T card : cards) {
        score += card.value();
     }
     return score;
   }

   public void addCard(T card) {
     cards.add(card);
   }
}
```

在上面的代码中，我们以泛型实现了Deck，同时把T的类型限定为Card。另外，我们还将Card实现成抽象类，这是因为如果不知道玩的是什么游戏，诸如value()的方法就没有太大意义。（你可能会据理力争，认为这些方法还是应该实现为好，以标准标准扑克牌规则实现默认值。）

现在，假设要构建二十一点游戏，我们需要知道这些牌的数值。人头牌K、Q、J等于10，Ace为11（大部分情况下为11，不过这应该交由Hand类负责，而不是交给下面这个类）。

```
public class BlackJackHand extends Hand<BlackJackCard> {
   /* 在二十一点玩法中，一手牌可以有多种分数，因为
    * Ace具有多个数值。若低于21就返回最高的分数，
    * 若高过21就返回最低的分数 */
   public int score() {
     ArrayList<Integer> scores = possibleScores();
     int maxUnder = Integer.MIN_VALUE;
     int minOver = Integer.MAX_VALUE;
     for (int score : scores) {
        if (score > 21 && score < minOver) {
          minOver = score;
        } else if (score <= 21 && score > maxUnder) {
          maxUnder = score;
        }
     }
     return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
   }

   /* 返回一个列表，包含这手牌所有可能的分数
    * （将Ace当作1和11进行计算） */
   private ArrayList<Integer> possibleScores() { ... }

   public boolean busted() { return score() > 21; }
   public boolean is21() { return score() == 21; }
   public boolean isBlackJack() { ... }
}

public class BlackJackCard extends Card {
   public BlackJackCard(int c, Suit s) { super(c, s); }
   public int value() {
     if (isAce()) return 1;
     else if (faceValue >= 11 && faceValue <= 13) return 10;
     else return faceValue;
   }

   public int minValue() {
     if (isAce()) return 1;
     else return value();
   }

   public int maxValue() {
     if (isAce()) return 11;
     else return value();
   }

   public boolean isAce() {
```

```
    return faceValue == 1;
  }

  public boolean isFaceCard() {
    return faceValue >= 11 && faceValue <= 13;
  }
}
```

这只是Ace的一种处理方式，另一种做法是创建一个继承自BlackJackCard的Ace类。

在本书所附、可下载的代码中，提供了一个可自动执行的二十一点游戏程序。

8.2　设想你有个呼叫中心，员工分成三个层级：接线员、主管和经理。客户来电会先分配给有空的接线员。若接线员处理不了，就必须将来电往上转给主管。若主管没空或是无法处理，则将来电往上转给经理。请设计这个问题的类和数据结构，并实现一个dispatchCall()方法，将客户来电分配给第一个有空的员工。（第 66 页）

解法

三个员工层级各有各的职责，因此，不同层级会有专门的函数。我们应该将它们放在各自对应的类里。

有些东西是所有员工都有的，比如地址、姓名、职位和年龄等。这些东西可以放在一个类里，再由其他类扩展或继承。

最后，还应该有一个CallHandler类，负责将来电分派给合适的负责人。

注意，任何面向对象设计问题，都会有很多不同的对象设计方式。请跟面试官讨论各种设计方案的优劣。通常，设计时应该从长远考虑，注重代码的灵活性和可维护性。

下面我们将详细说明每个类。

CallHandler实现为一个单态类，它是程序的主体，所有来电都先由这个类进行分派。

```
public class CallHandler {
  private static CallHandler instance;

  /* 三个员工层级：接线员、主管、经理 */
  private final int LEVELS = 3;

  /* 起始设定10位接线员、4位主管和2位经理 */
  private final int NUM_RESPONDENTS = 10;
  private final int NUM_MANAGERS = 4;
  private final int NUM_DIRECTORS = 2;

  /* 员工列表，以层级区分：
   * employeeLevels[0] = 接线员
   * employeeLevels[1] = 主管
   * employeeLevels[2] = 经理
   */
  List<List<Employee>> employeeLevels;

  /* 存放来电层级的队列 */
  List<List<Call>> callQueues;
```

```

    protected  CallHandler() { ... }

    /* 取得单态类的实例 */
    public static CallHandler getInstance() {
      if (instance == null) instance = new CallHandler();
      return instance;
    }

    /* 找出第一个有空处理来电的员工 */
    public Employee getHandlerForCall(Call call) { ... }

    /* 将来电分派给有空的员工，若没人有空，
     * 就存放在队列中 */
    public void dispatchCall(Caller caller) {
      Call call = new Call(caller);
      dispatchCall(call);
    }

    /* 将来电分配给有空的员工，若没人有空，
     * 就存放在队列中 */
    public void dispatchCall(Call call) {
      /* 试着将来电分派给层级最低的员工 */
      Employee emp = getHandlerForCall(call);
      if (emp != null) {
         emp.receiveCall(call);
         call.setHandler(emp);
      } else {
         /* 根据来电级别，将来电放到相应的
          * 队列中 */
         call.reply("Please wait for free employee to reply");
         callQueues[call.getRank().getValue()].add(call);
      }
    }

    /* 有员工有空了，查找该员工可服务的来电。
     * 若分派了来电则返回true，否则返回false */
    public boolean assignCall(Employee emp) { ... }
}
```

`Call`代表客户来电，每次来电会有个最低层级，并且会被分派给第一个可处理该来电的员工。

```
public class Call {
    /* 可处理此来电的最低层级员工 */
    private Rank rank;

    /* 拨号方 */
    private Caller caller;

    /* 处理来电的员工 */
    private Employee handler;

    public Call(Caller c) {
      rank = Rank.Responder;
```

```
    caller = c;
  }

  /* 设定处理来电的员工 */
  public void setHandler(Employee e) { handler = e; }

  public void reply(String message) { ... }
  public Rank getRank() { return rank; }
  public void setRank(Rank r) { rank = r; }
  public Rank incrementRank() { ... }
  public void disconnect() { ... }
}
```

Employee是Director、Manager和Respondent类的父类，由于没有必要直接实例化Employee类，因此是个抽象类。

```
abstract class Employee {
  private Call currentCall = null;
  protected Rank rank;

  public Employee() { }

  /* 开始交谈对话 */
  public void receiveCall(Call call) { ... }

  /* 问题解决了，结束来电 */
  public void callCompleted() { ... }

  /* 问题未解决，往上转给更高层级的员工，
   * 并为该员工分派新的来电 */
  public void escalateAndReassign() { ... }
  }

  /* 分派新的来电给该员工，若他有空的话 */
  public boolean assignNewCall() { ... }

  /* 返回该员工是否有空 */
  public boolean isFree() { return currentCall == null; }

  public Rank getRank() { return rank; }
}

```

有了Employee类，Respondent、Director和Manager只是在此基础上稍微扩展一下。

```
class Director extends Employee {
  public Director() {
    rank = Rank.Director;
  }
}

class Manager extends Employee {
  public Manager() {
    rank = Rank.Manager;
```

```
    }
}

class Respondent extends Employee {
    public Respondent() {
      rank = Rank.Responder;
    }
}
```

上面只是此题的一种设计方式。注意，其实还有其他许多同样不错的方法。

在面试中，要写这么多代码似乎有点可怕，确实如此。这里给出的代码比较完整，在实际面试中，可能不需要写得这么全，有些细节可以先简略带过，等到有时间了再作补充。

8.3　运用面向对象原则，设计一款音乐点唱机。（第 66 页）

解法

但凡遇到面向对象设计的问题，一开始就要向面试官问几个问题，以便厘清设计时有哪些限制条件。这台点唱机放的是CD吗？是唱片？还是MP3？它是计算机模拟软件，还是代表一台实体点唱机？播放音乐要收钱还是免费？收钱的话，要求哪国货币？可以找零吗？

遗憾的是，这里没有面试官，我们无法与之对话。因此，下面将作出一些假设。假设这台点唱机为计算机模拟软件，与实体点唱机非常相像，另外，假定播放音乐是免费的。

至此尘埃落定，下面将列出基本的系统组件：

- 点唱机（Jukebox）;
- CD;
- 歌曲（Song）;
- 艺术家（Artist）;
- 播放列表（Playlist）;
- 显示屏（Display，在屏幕上显示详细信息）。

接下来，进一步分解上述组件，考虑可能的动作。

- 新建播放列表（包括新增、删除和随机播放）
- CD选择器
- 歌曲选择器
- 将歌曲放进播放队列
- 获取播放列表中的下一首歌曲

另外，还可引入用户：

- 添加;
- 删除;
- 信用信息。

每个主要系统组件大致都会转换成一个对象，而每个动作则转换为一个方法。下面将介绍一种可行的设计。

Jukebox类代表此题的主体，系统各个组件之间或系统与用户间的大量交互，都是通过这个类实现的。

```
public class Jukebox {
   private CDPlayer cdPlayer;
   private User user;
   private Set<CD> cdCollection;
   private SongSelector ts;

   public Jukebox(CDPlayer cdPlayer, User user,
                  Set<CD> cdCollection, SongSelector ts) {
     ...
   }

   public Song getCurrentSong() {
     return ts.getCurrentSong();
   }

   public void setUser(User u) {
     this.user = u;
   }
}
```

跟实际CD播放器一样，CDPlayer类一次只能放一张CD。不在播放的CD都存放在点唱机里。

```
public class CDPlayer {
   private Playlist p;
   private CD c;

   /* 构造函数 */
   public CDPlayer(CD c, Playlist p) { ... }
   public CDPlayer(Playlist p) { this.p = p; }
   public CDPlayer(CD c) { this.c = c; }

   /* 播放歌曲 */
   public void playSong(Song s) { ... }

   /* getter和setter */
   public Playlist getPlaylist() { return p; }
   public void setPlaylist(Playlist p) { this.p = p; }

   public CD getCD() { return c; }
   public void setCD(CD c) { this.c = c; }
}
```

Playlist类管理当前播放的歌曲和待播放的下一首歌曲。它本质上是播放队列的包裹类，还提供了一些操作起来更方便的方法。

```
public class Playlist {
   private Song song;
   private Queue<Song> queue;
   public Playlist(Song song, Queue<Song> queue) {
     ...
   }
```

```
    public Song getNextSToPlay() {
      return queue.peek();
    }
    public void queueUpSong(Song s) {
      queue.add(s);
    }
}
```

CD、Song和User这几个类都相当简单，主要由成员变量、getter（访问）和setter（设置）方法组成。

```
public class CD {
    /* 识别码、艺术家、歌曲等 */
}

public class Song {
    /* 识别码、CD（可能为空）、名称、长度等 */
}

public class User {
    private String name;
    public String getName() { return name; }
    public void setName(String name) {  this.name = name; }
    public long getID() { return ID; }
    public void setID(long iD) { ID = iD; }
    private long ID;
    public User(String name, long iD) { ... }
    public User getUser() { return this; }
    public static User addUser(String name, long iD) { ... }
}
```

这当然绝非唯一“正确”的实现。跟其他限制条件一样，面试官对一开始询问的回应也会影响点唱机里各种类的设计。

8.4　运用面向对象原则，设计一个停车场。（第 66 页）

解法

这个问题的表述有些含糊，在实际的面试中也会出现这种情况。这就要求你与面试官交流，问清楚允许哪些车辆进入停车场，它是不是多层的，等等。

为便于描述，我们先做如下假设条件。这些特定的假设条件会让问题变得更复杂，但又不致过于复杂。如果你想作出其他假设，那也完全不成问题。

- 停车场是多层的。每一层有好几排停车位。
- 停车场可停放摩托车、轿车和大巴。
- 停车场有摩托车车位、小车位和大车位。
- 摩托车可停在任意车位上。
- 轿车可停在单个小车位或大车位上。
- 大巴可停在同一排五个连续的大车位上，但不能停在小车位上。

在下面的实现中，我们创建了抽象类Vehicle，而Car、Bus和Motorcycle都继承自这个类。为处理不同大小的车位，我们用了一个类ParkingSpot，并以它的成员变量表示车位大小。

```
public enum VehicleSize { Motorcycle, Compact, Large }

public abstract class Vehicle {
   protected ArrayList<ParkingSpot> parkingSpots =
     new ArrayList<ParkingSpot>();
   protected String licensePlate;
   protected int spotsNeeded;
   protected VehicleSize size;

   public int getSpotsNeeded() { return spotsNeeded; }
   public VehicleSize getSize() { return size; }

   /* 将车辆停在这个车位里（也可能包含其他车位） */
   public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }

   /* 从车位移除车辆，并通知车位车辆已离开 */
   public void clearSpots() { ... }

   /* 检查车位是否够大以停放该车辆（且车位是空的），
    * 这只会检查车位大小，并不检查是否有足够多
    * 的车位 */
   public abstract boolean canFitInSpot(ParkingSpot spot);
}

public class Bus extends Vehicle {
   public Bus() {
     spotsNeeded = 5;
     size = VehicleSize.Large;
   }

   /* 检查车位是否为大车位，不会检查车位的数目 */
   public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Car extends Vehicle {
   public Car() {
     spotsNeeded = 1;
     size = VehicleSize.Compact;
   }

   /* 检查车位是小车位还是大车位 */
   public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Motorcycle extends Vehicle {
   public Motorcycle() {
     spotsNeeded = 1;
     size = VehicleSize.Motorcycle;
   }

```

9

```
    public boolean canFitInSpot(ParkingSpot spot) { ... }
}
```

ParkingLot类本质上就是Level数组的包裹类。以这种方式实现，我们就能将真正寻找空车位和泊车的处理逻辑从ParkingLot里更为广泛的动作中抽取出来。要是不这么做，就需要将车位放在某种双数组中（或将车位位于所在楼层的编号对应到车位列表的散列表）。将ParkingLot与Level分离开来，整个设计更显清晰。

```
public class ParkingLot {
   private Level[] levels;
   private final int NUM_LEVELS = 5;

   public ParkingLot() { ... }

   /* 将该车辆停在一个车位或多个车位，
    * 失败则返回false */
   public boolean parkVehicle(Vehicle vehicle) { ... }
}

/* 代表停车场里的一层 */
public class Level {
   private int floor;
   private ParkingSpot[] spots;
   private int availableSpots = 0; // 空闲车位的数量
   private static final int SPOTS_PER_ROW = 10;

   public Level(int flr, int numberSpots) { ... }

   public int availableSpots() { return availableSpots; }

   /* 找地方停这辆车，失败则返回false */
   public boolean parkVehicle(Vehicle vehicle) { ... }

   /* 停放该车辆，从车位编号spotNumber开始，
    * 直到vehicle.spotsNeeded */
   private boolean parkStartingAtSpot(int num, Vehicle v) { ... }

   /* 寻找车位停放这辆车。返回车位索引号，
    * 失败则返回-1 */
   private int findAvailableSpots(Vehicle vehicle) { ... }

   /* 当有车辆从车位移除时，增加可用车位数
    * availableSpots */
   public void spotFreed() { availableSpots++; }
}
```

ParkingSpot类只用一个变量表示车位的大小。我们也可以从ParkingSpot继承并创建LargeSpot、CompactSpot和MotorcycleSpot等几个类来实现，但这么做未免有些小题大做。除了大小不一，这些车位并没有不一样的行为。

```
public class ParkingSpot {
   private Vehicle vehicle;
```

```
    private VehicleSize spotSize;
    private int row;
    private int spotNumber;
    private Level level;

    public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}

    public boolean isAvailable() { return vehicle == null; }

    /* 检查车位是否够大、可用 */
    public boolean canFitVehicle(Vehicle vehicle) { ... }

    /* 将车辆停在该车位 */
    public boolean park(Vehicle v) { ... }

    public int getRow() { return row; }
    public int getSpotNumber() { return spotNumber; }

    /* 从车位移除车辆，并通知楼层，
     * 有新的车位可用 */
    public void removeVehicle() { ... }
}
```

在本书可下载的源码包中，可以找到上述代码的完整实现，包括可执行的测试代码。

8.5 请设计在线图书阅读器系统的数据结构。（第 66 页）

解法

此题对系统功能的说明着墨不多，因此，就让我们假设要设计一个基本的在线图书阅读系统，提供如下功能。

- 用户成员资格的建立和延长期限。
- 搜索图书数据库。
- 阅读书籍。
- 同一时间只能有一个活跃用户。
- 该用户一次只能看一本书。

要实现这些操作，可能还需提供许多其他函数，比如get、set、update，等等。该系统的对象可能包括User、Book和Library。

OnlineReaderSystem类为程序的主体，可以这么实现：存放所有图书的信息，管理用户，刷新显示画面，但是这么一来，整个类就会变得非常笨重。因此，我们转而选择将这些组件拆分成Library、UserManager和Display等几个类。

```
public class OnlineReaderSystem {
    private Library library;
    private UserManager userManager;
    private Display display;

    private Book activeBook;
    private User activeUser;
```

9

```

  public OnlineReaderSystem() {
    userManager = new UserManager();
    library = new Library();
    display = new Display();
  }

  public Library getLibrary() { return library;  }
  public UserManager getUserManager() { return userManager; }
  public Display getDisplay() { return display; }

  public Book getActiveBook() { return activeBook; }
  public void setActiveBook(Book book) {
    activeBook = book;
    display.displayBook(book);
  }

  public User getActiveUser() { return activeUser; }
  public void setActiveUser(User user) {
    activeUser = user;
    display.displayUser(user);
  }
}
```

随后，我们实现这几个类，以处理用户管理器、图书库和显示组件。

```
public class Library {
  private Hashtable<Integer, Book> books;

  public Book addBook(int id, String details) {
    if (books.containsKey(id)) {
      return null;
    }
    Book book = new Book(id, details);
    books.put(id, book);
    return book;
  }

  public boolean remove(Book b) { return remove(b.getID()); }
  public boolean remove(int id) {
    if (!books.containsKey(id)) {
       return false;
    }
    books.remove(id);
    return true;
  }

  public Book find(int id) {
    return books.get(id);
  }
}

public class UserManager {
  private Hashtable<Integer, User> users;
```

```

    public User addUser(int id, String details, int accountType) {
      if (users.containsKey(id)) {
         return null;
      }
      User user = new User(id, details, accountType);
      users.put(id, user);
      return user;
    }

    public boolean remove(User u) {
      return remove(u.getID());
    }

    public boolean remove(int id) {
      if (!users.containsKey(id)) {
         return false;
      }
      users.remove(id);
      return true;
    }

    public User find(int id) {
      return users.get(id);
    }
}

public class Display {
    private Book activeBook;
    private User activeUser;
    private int pageNumber = 0;

    public void displayUser(User user) {
      activeUser = user;
      refreshUsername();
    }

    public void displayBook(Book book) {
      pageNumber = 0;
      activeBook = book;

      refreshTitle();
      refreshDetails();
      refreshPage();
    }

    public void turnPageForward() {
      pageNumber++;
      refreshPage();
    }

    public void turnPageBackward() {
      pageNumber--;
      refreshPage();
```

```
  }

  public void refreshUsername() { /* 更新用户名的显示 */ }
  public void refreshTitle() { /* 更新标题的显示 */ }
  public void refreshDetails() { /* 更新细节信息的显示 */  }
  public void refreshPage() { /* 更新页面显示 */ }
}
```

User和Book类只是存放数据，并没有什么真正的功能。

```
public class Book {
  private int bookId;
  private String details;

  public Book(int id, String det) {
    bookId = id;
    details = det;
  }

  public int getID() { return bookId;  }
  public void setID(int id) { bookId = id; }
  public String getDetails() { return details; }
  public void setDetails(String d) { details = d; }
}

public class User {
  private int userId;
  private String details;
  private int accountType;

  public void renewMembership() {  }

  public User(int id, String details, int accountType) {
    userId = id;
    this.details = details;
    this.accountType = accountType;
  }

  /* getter和setter */
  public int getID() { return userId; }
  public void setID(int id) { userId = id; }
  public String getDetails() {
    return details;
  }

  public void setDetails(String details) {
    this.details = details;
  }
  public int getAccountType() { return accountType; }
  public void setAccountType(int t) { accountType = t; }
}
```

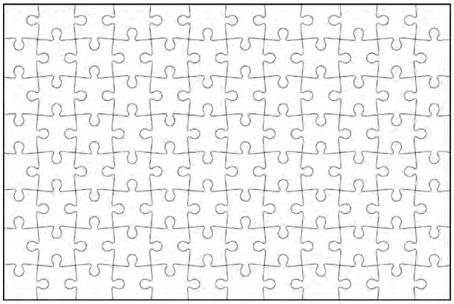
用户管理、图书库和显示功能等功能本可以通通放进OnlineReaderSystem类中，这里却将它们拆分至不同的类里，这么做挺有意思的，值得探讨一番。如果一个系统很小，这么做可能会

使系统变得过于复杂。然而，随着系统的扩展，OnlineReaderSystem会加入越来越多的功能，将各个功能拆分开来，可以避免这个主类变得臃肿不堪。

8.6　实现一个拼图程序。设计相关数据结构并提供一种拼图算法。假设你有一个fitsWith方法，传入两块拼图，若两块拼图能拼在一起，则返回true。（第 66 页）

解法

假设有一套传统简单的拼图游戏，按行和列划分为网格，每块拼图都落在某一行和某一列中，有四条边，每条边分为三种：内凹、外凸和平直。例如，角落的拼图块有两条边是平直的，另外两条边可能是内凹或外凸。



在玩拼图游戏时（手动或借助算法），我们需要存储每块拼图的位置，位置可以是绝对的或相对的。

- **绝对位置**：“这块拼图的位置是(12, 23)。”绝对位置属于Piece类本身，同时还包含摆放方向。
- **相对位置**：“我不知道这块拼图的实际位置，但知道它与另一块拼图相邻。”相对位置属于Edge类。

我们的解法只使用相对位置，从而将相邻的边拼在一起。

下面是一种可能的面向对象设计：

```
class Edge {
   enum Type { inner, outer, flat }
   Piece parent;
   Type type;
   int index; // 指向Piece.edges的索引
   Edge attached_to; // 相对位置

   /* 参见算法一节，若两块拼图应该拼在一起，
    * 则返回true */
   boolean fitsWith(Edge edge) { ... };
}

class Piece {
   Edge[] edges;
   boolean isCorner() { ... }
```

```
}

class Puzzle {
   Piece[] pieces; /* 剩余还未拼的拼图 */
   Piece[][] solution;

   /* 参见算法一节 */
   Edge[] inners, outers, flats;
   Piece[] corners;

   /* 参见算法一节 */
   void sort() { ... }
   void solve() { ...}
}
```

拼拼图的算法

下面我们将搭配使用伪码和实际代码，勾勒出拼拼图的算法。

就跟小孩玩拼图游戏时一样，我们会从最简单的拼图块入手：四个角落和四条边上的。我们很容易就能从所有拼图块中找出直边的。拼图的时候，不妨将拼图块按边缘类型分组，这或许是个不错的选择。

```
void sort() {
   for each Piece p in pieces {
     if (p has two flat edges) then add p to corners
     for each edge in p.edges {
       if edge is inner then add to inners
       if edge is outer then add to outers
     }
   }
}
```

如此一来，给定某一边，我们可以更快速地挑出可能拼合在一起的拼图块。然后一行一行地检查拼图，找出可拼在一起的拼图块。

下面实现的`solve`方法会随意挑选一个角落开始拼图，找出这个角落还没拼上的一边，然后试着找出可拼在一起的拼图块。找到相符的拼图块以后，执行如下操作。

(1) 与边缘衔接起来。

(2) 从未接好的边缘列表中移除该边缘。

(3) 找到下一条未接好的边缘。

如果当前边缘的对边还未接好，则下一条未接好的边缘即为该边缘。如果该边缘已接好，则下一条边可以是任意其他边缘。这会让拼拼图时看起来像是从外向内的螺旋状。

呈螺旋状的原因是，只要可以的话，该算法总是以直线移动。当抵达第一边缘的末端时，算法会移至角落拼图块唯一可用的边缘，也就是旋转90度。每到边缘的末端就会旋转90度，直到拼图外圈边缘全部拼完。当最后一块边缘的拼图块拼好后，该拼图块只剩一条边没接好，于是再次旋转90度。在后续每一圈中，该算法会重复同样的流程，直至所有拼图块都拼好为止。

下面是该算法的类似Java的伪码实现。

```
public void solve() {
  /* 随便选个角落开始拼图 */
  Edge currentEdge = getExposedEdge(corner[0]);

  /* 循环会以螺旋状进行迭代，
   * 直到拼图完成为止 */
  while (currentEdge != null) {
    /* 以相反的边缘类型进行拼图，内凹对外凸，等等 */
    Edge[] opposites = currentEdge.type == inner ?
                       outers : inners;
    for each Edge fittingEdge in opposites {
      if (currentEdge.fitsWith(fittingEdge)) {
        attachEdges(currentEdge, fittingEdge); // 衔接边缘
        removeFromList(currentEdge);
        removeFromList(fittingEdge);

        /* 取出下一条边缘 */
        currentEdge = nextExposedEdge(fittingEdge);
        break; // 跳出内层循环，继续外层循环
      }
    }
  }
}

public void removeFromList(Edge edge) {
  if (edge.type == flat) return;
  Edge[] array = currentEdge.type == inner ? inners : outers;
  array.remove(edge);
}

/* 可以的话，返回对边的边缘，否则，
 * 返回任意还未接好的边缘 */
public Edge nextExposedEdge(Edge edge) {
  int next_index = (edge.index + 2) % 4; // 对边
  Edge next_edge = edge.parent.edges[next_index];
  if isExposed(next_edge) {
    return next_edge;
  }
  return getExposedEdge(edge.parent);
}

public Edge attachEdges(Edge e1, Edge e2) {
  e1.attached_to = e2;
  e2.attached_to = e1;
}

public Edge isExposed(Edge e1) {
  return edge.type != flat && edge.attached_to == null;
}

public Edge getExposedEdge(Piece p) {
  for each Edge edge in p.edges {
    if (isExposed(edge)) {
      return edge;
```

```
        }
    }
    return null;
}
```

为了简单起见，我们将inners和outers表示为一个Edge数组。但这并不是个好设计，因为需要频繁添加和删除数组元素。在实际代码开发中，我们可能会用链表来实现这些变量。

对面试来说，要写出此题的完整代码，实在太多了。通常，面试官可能只会要求你勾勒代码的轮廓。

8.7　请描述该如何设计一个聊天服务器。要求给出各种后台组件、类和方法的细节，并说明其中最难解决的问题会是什么。（第 66 页）

解法

设计聊天服务器是项大工程，绝非一次面试就能完成。毕竟，就算一整个团队，也要花费数月乃至好几年才能打造出一个聊天服务器。作为求职者，你的工作是专注解决该问题的某个方面，涉及范围要够广，又要够集中，这样才能在一轮面试中搞定。它不一定要与真实情况一模一样，但也应该忠实反映出实际的实现。

这里我们会把注意力放在用户管理和对话等核心功能：添加用户、创建对话、更新状态，等等。考虑到时间和空间有限，我们不会探讨这个问题的联网部分，也不描述数据是怎么真正推送到客户端的。

另外，我们假设“好友关系”是双向的，如果你是我的联系人之一，那就表示我也是你的联系人之一。我们的聊天系统将支持群组聊天和一对一（私密）聊天，但并不考虑语音聊天、视频聊天或文件传输。

1. 需要支持哪些特定动作？

这也有待你跟面试官探讨，下面列出几点想法。

- 显示在线和离线状态。
- 添加请求（发送、接受、拒绝）。
- 更新状态信息。
- 发起私聊和群聊。
- 在私聊和群聊中添加新信息。

这只是一部分列表，如果时间有富余，还可以多加一些动作。

2. 从这些需求可了解到什么？

我们必须掌握用户、添加请求的状态、在线状态和消息等概念。

3. 系统有哪些核心组件？

这个系统可能由一个数据库、一组客户端和一组服务器组成。我们的面向对象设计不会包含这些部分，不过可以讨论一下系统的整体概览。

数据库将用来存放更持久的数据，比如用户列表或聊天对话的备份。SQL数据库应该是不错

的选择，或者，如果可扩展性要求更高，可以选用BigTable或其他类似的系统。

对于客户端和服务器之间的通信，使用XML应该也不错。尽管这种格式不是最紧凑的（你也应该向面试官指出这一点），它仍是很不错的选择，因为不管是计算机还是人类都容易辨识。使用XML可以让程序调试起来更轻松，这一点非常重要。

服务器由一组机器组成，数据会分散到各台机器上，这样一来，我们可能就必须从一台机器跳到另一台机器。如果可能的话，我们会尽量在所有机器上复制部分数据，以减少查询操作的次数。在此，设计上有个重要的限制条件，就是必须防止出现单点故障。例如，如果一台机器控制所有用户的登录，那么，只要这一台机器断网，就会造成数以百万计的用户无法登录。

4. 有哪些关键的对象和方法？

系统的关键对象包括用户、对话和状态消息等，我们已经实现了UserManagement类。要是更关注这个问题的联网方面或其他组件，我们就可能转而深入探究那些对象。

```
/* UserManager用作核心用户动作的控制中心 */
public class UserManager {
   private static UserManager instance;
   /* 从用户识别码映射到用户 */
   private HashMap<Integer, User> usersById;

   /* 从帐户名映射到用户 */
   private HashMap<String, User> usersByAccountName;

  /* 从用户识别码映射到在线用户 */
   private HashMap<Integer, User> onlineUsers;

   public static UserManager getInstance() {
     if (instance == null) instance = new UserManager();
     return instance;
   }

   public void addUser(User fromUser, String toAccountName) { ... }
   public void approveAddRequest(AddRequest req) { ... }
   public void rejectAddRequest(AddRequest req) { ... }
   public void userSignedOn(String accountName) { ... }
   public void userSignedOff(String accountName) { ... }
}
```

在User类中，receivedAddRequest方法会通知用户B（User B），用户A（User A）请求加他为好友。用户B会接受或拒绝该请求（通过UserManager.approveAddRequest或rejectAddRequest），UserManager则负责将用户互相添加到对方的通讯录中。

当UserManager要将AddRequest加入用户A的请求列表时，会调用User类的sentAddRequest方法。综上，整个流程如下。

(1) 用户A点击客户端软件上的“添加用户”，发送给服务器。

(2) 用户A调用requestAddUser(User B)。

(3) 步骤2的方法会调用UserManager.addUser。

(4) UserManager会调用User A.sentAddRequest和User B.receivedAddRequest。

重申一下，这只是设计这些交互的其中一种方式。但这不是唯一的方式，甚至也不是唯一“好”的做法。

```
public class User {
   private int id;
   private UserStatus status = null;

   /* 将其他参与的用户识别码映射到对话 */
   private HashMap<Integer, PrivateChat> privateChats;

   /* 将群聊识别码映射到群聊 */
   private ArrayList<GroupChat> groupChats;

   /* 将其他人的用户识别码映射到加入请求 */
   private HashMap<Integer, AddRequest> receivedAddRequests;

   /* 将其他人的用户识别码映射到加入请求 */
   private HashMap<Integer, AddRequest> sentAddRequests;

   /* 将用户识别码映射到加入请求 */
   private HashMap<Integer, User> contacts;

   private String accountName;
   private String fullName;

   public User(int id, String accountName, String fullName) { ... }
   public boolean sendMessageToUser(User to, String content){ ... }
   public boolean sendMessageToGroupChat(int id, String cnt){...}
   public void setStatus(UserStatus status) { ... }
   public UserStatus getStatus() { ... }
   public boolean addContact(User user) { ... }
   public void receivedAddRequest(AddRequest req) { ...}
   public void sentAddRequest(AddRequest req) { ... }
   public void removeAddRequest(AddRequest req) { ... }
   public void requestAddUser(String accountName) { ... }
   public void addConversation(PrivateChat conversation) { ... }
   public void addConversation(GroupChat conversation) { ... }
   public int getId() { ... }
   public String getAccountName() { ... }
   public String getFullName() { ... }
}
```

Conversation 类实现为一个抽象类，因为所有 Conversation 不是 GroupChat 就是 PrivateChat，同时每个类各有自己的功能。

```
public abstract class Conversation {
   protected ArrayList<User> participants;
   protected int id;
   protected ArrayList<Message> messages;

   public ArrayList<Message> getMessages() { ... }
   public boolean addMessage(Message m) { ... }
   public int getId() { ... }
}

```

```java
public class GroupChat extends Conversation {
   public void removeParticipant(User user) { ... }
   public void addParticipant(User user) { ... }
}

public class PrivateChat extends Conversation {
   public PrivateChat(User user1, User user2) { ...
   public User getOtherParticipant(User primary) { ... }
}

public class Message {
   private String content;
   private Date date;
   public Message(String content, Date date) { ... }
   public String getContent() { ... }
   public Date getDate() { ... }
}
```

AddRequest和UserStatus两个类比较简单，功能不多，主要用来将数据聚合在一起，方便其他类使用。

```java
public class AddRequest {
   private User fromUser;
   private User toUser;
   private Date date;
   RequestStatus status;

   public AddRequest(User from, User to, Date date) { ... }
   public RequestStatus getStatus() { ... }
   public User getFromUser() { ... }
   public User getToUser() { ... }
   public Date getDate() { ... }
}

public class UserStatus {
    private String message;
    private UserStatusType type;
    public UserStatus(UserStatusType type, String message) { ... }
    public UserStatusType getStatusType() { ... }
    public String getMessage() { ... }
}

public enum UserStatusType {
   Offline, Away, Idle, Available, Busy
}

public enum RequestStatus {
   Unread, Read, Accepted, Rejected
}
```

在本书可下载的完整源码中，可以查看这些方法的更多细节，包括上述方法的具体实现。

5. 最难解决或最有意思的问题是什么？

下面这些问题可能有点意思，不妨与面试官深入探讨一番。

问题 1：如何确定某人在线——我指的是真的、真的知道？

虽然希望用户在退出时通知我们，但即便如此也无法确切知道状态。例如，用户的网络连接可能断开了。为了确定用户何时退出，或许可以试着定期询问客户端，以确保它仍然在线。

问题 2：如何处理冲突的信息？

部分信息存储在计算机内存中，部分则存储在数据库里。如果两者不同步有冲突，那会出什么问题？哪一部分是“正确的”？

问题 3：如何才能让服务器在任何负载下都能应付自如？

前面我们设计聊天服务器时并没怎么考虑可扩展性，但在实际场景中必须予以关注。我们需要将数据分散到多台服务器上，而这又要求我们更关注数据的不同步。

问题 4：如何预防拒绝服务攻击？

客户端可以向我们推送数据——若它们试图向服务器发起拒绝服务（DOS）攻击，怎么办？该如何预防？

8.8　“奥赛罗棋”（黑白棋）的玩法如下：每一枚棋子的一面为白，一面为黑。游戏双方各执黑、白棋子对决，当一枚棋子的左右或上下同时被对方棋子夹住，这枚棋子就算是被吃掉了，随即翻面为对方棋子的颜色。轮到你落子时，必须至少吃掉对方一枚棋子。任意一方无子可落时，游戏即告结束。最后，棋盘上棋子较多的一方获胜。请运用面向对象设计方法，实现“奥赛罗棋”。（第 66 页）

解法

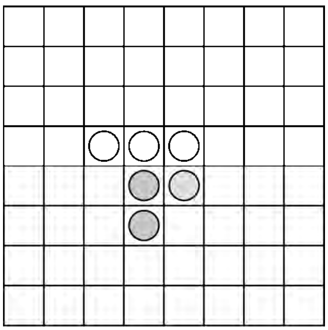
我们先来举个例子。假设在一盘奥赛罗棋中，有如下棋步。

(1) 初始化棋盘，在中心位置布下两枚黑子和两枚白子。两枚黑子分别落在中心点的左上方和右下方。

(2) 在6行4列处落黑子，则5行4列的白子翻面变为黑子。

(3) 在4行3列处落白子，则4行4列的黑子翻面变为白子。

经过上面的棋步，棋盘布局如下。



在奥赛罗棋中，核心对象大致有游戏（game）、棋盘（board）、棋子（piece，黑子或白子）和玩家（player）。该如何用面向对象设计优雅地表示这些对象？

1. 该不该创建 BlackPiece 和 WhitePiece 类？

起先，我们可能认为自己需要从Piece抽象类派生出BlackPiece类和WhitePiece类。然而，这么做不见得好。每颗棋子都可以来回翻面，黑变白，白变黑，这么来看，连续不断地销毁和创建完全相同的对象并不明智。因此，更好的做法可能是只创建Piece类，并用标记指示棋子当前的颜色。

2. 需要 Board 和 Game 两个独立的类吗？

严格来说，可能没有必要既创建Game对象又引入Board对象。不过，分别创建这两个对象可以从逻辑上划分棋盘（只含涉及落子的逻辑处理）和游戏（含计时、游戏流程等）。但是这么做也有弊端，我们的程序会多加几层处理，变得更复杂。有个函数可能会调用Game的方法，却只是为了让它去调用Board里的方法。下面我们决定将Game和Board分开创建，不过面试时最好跟面试官讨论一下。

3. 谁来记录分数？

很显然，我们需要某种记分方式来记录黑子和白子的数目。但该由程序的哪部分来负责维护这些信息？不管是由Game抑或Board甚至由Piece（在静态方法中）维护这些信息，各有各的理由。我们选择交由Board保存这部分信息，分数在逻辑上可以算是棋盘的一部分，由Piece或Board调用Board类的colorChanged和colorAdded方法进行更新。

4. Game 该不该实现成单态类？

将Game实现为单态类，优点在于Game的方法调用起来很容易，不用将Game对象的引用传来传去。

不过，将Game实现成单态类也意味着它只能实例化一次，这个假设条件成立吗？在面试时，最好与面试官交流一下。

下面是奥赛罗棋的一种可能设计。

```
public enum Direction {
   left, right, up, down
}

public enum Color {
   White, Black
}

public class Game {
   private Player[] players;
   private static Game instance;
   private Board board;
   private final int ROWS = 10;
   private final int COLUMNS = 10;
```

```java

  private Game() {
    board = new Board(ROWS, COLUMNS);
    players = new Player[2];
    players[0] = new Player(Color.Black);
    players[1] = new Player(Color.White);
  }

  public static Game getInstance() {
    if (instance == null) instance = new Game();
    return instance;
  }

  public Board getBoard() {
    return board;
  }
}
```

Board类负责管理棋子本身，但并不处理游戏玩法的部分，而是交由Game类处理。

```java
public class Board {
  private int blackCount = 0;
  private int whiteCount = 0;
  private Piece[][] board;

  public Board(int rows, int columns) {
    board = new Piece[rows][columns];
  }

  public void initialize() {
    /* 初始化棋盘中心的白子和黑子 */
  }

  /* 试着将颜色为color的棋子放在(row, column)位置
   * 成功则返回true */
  public boolean placeColor(int row, int column, Color color) {
    ...
  }

  /* 从(row, column)开始，顺着方向d，
   * 将棋子翻面 */
  private int flipSection(int row, int column, Color color,
                          Direction d) { ... }

  public int getScoreForColor(Color c) {
    if (c == Color.Black) return blackCount;
    else return whiteCount;
  }

  /* 更新棋盘，有newPieces个棋子变为newColor颜色，
   * 减少另一种颜色的分数 */
  public void updateScore(Color newColor, int newPieces) { ... }
}
```

如前所述，我们会用Piece类实现黑白棋子，该类有个简单的Color变量，表示棋子是黑子还是白子。

```
public class Piece {
   private Color color;
   public Piece(Color c) { color = c; }

   public void flip() {
     if (color == Color.Black) color = Color.White;
     else color = Color.Black;
   }

   public Color getColor() { return color; }
}
```

Player存放的信息非常有限，甚至不会保存自己的分数，但有个方法可用来获取分数。Player.getScore()会调用GameManager取得分数。

```
public class Player {
   private Color color;
   public Player(Color c) { color = c; }

   public int getScore() { ...  }

   public boolean playPiece(int r, int c) {
     return Game.getInstance().getBoard().placeColor(r, c, color);
   }

   public Color getColor() { return color; }
}
```

本书可下载的源码包提供了完整可运行的版本。

记住，在处理很多问题时，相比你做了些什么，你为什么这么做反而更显重要。面试官也许不会在意你是否选择将Game类实现为单态类，但她可能真的在乎你有没有花时间思考，有没有跟她讨论各种做法的优劣。

8.9 设计一种内存文件系统（in-memory file system）的数据结构和算法，并说明具体做法。如有可行，请用代码举例说明。（第 66 页）

解法

许多求职者一看到这个问题，可能就会惊慌失措。文件系统太底层了吧！

其实，没必要惊慌。只要把文件系统的组件考虑周全，我们就能像解决其他面向对象设计问题那样搞定此题。

一个最简单的文件系统由File（文件）和Directory（目录）组成。每个Directory包含一组File和Directory。File和Directory有很多相同特征，因此我们创建了Entry类，前面两个类则继承这个类。

```
public abstract class Entry {
```

```
  protected Directory parent;
  protected long created;
  protected long lastUpdated;
  protected long lastAccessed;
  protected String name;

  public Entry(String n, Directory p) {
    name = n;
    parent = p;
    created = System.currentTimeMillis();
    lastUpdated = System.currentTimeMillis();
    lastAccessed = System.currentTimeMillis();
  }

  public boolean delete() {
    if (parent == null) return false;
    return parent.deleteEntry(this);
  }

  public abstract int size();

  public String getFullPath() {
    if (parent == null) return name;
    else return parent.getFullPath() + "/" + name;
  }

  /* getter和setter */
  public long getCreationTime() { return created; }
  public long getLastUpdatedTime() { return lastUpdated; }
  public long getLastAccessedTime() { return lastAccessed; }
  public void changeName(String n) { name = n; }
  public String getName() { return name; }
}

public class File extends Entry {
  private String content;
  private int size;

  public File(String n, Directory p, int sz) {
    super(n, p);
    size = sz;
  }

  public int size() { return size; }
  public String getContents() { return content; }
  public void setContents(String c) { content = c; }
}

public class Directory extends Entry {
  protected ArrayList<Entry> contents;

  public Directory(String n, Directory p) {
    super(n, p);
    contents = new ArrayList<Entry>();
```

```
  }

  public int size() {
    int size = 0;
    for (Entry e : contents) {
       size += e.size();
    }
    return size;
  }

  public int numberOfFiles() {
    int count = 0;
    for (Entry e : contents) {
       if (e instanceof Directory) {
         count++; // 目录也算作文件
         Directory d = (Directory) e;
         count += d.numberOfFiles();
       } else if (e instanceof File) {
         count++;
       }
    }
    return count;
  }

  public boolean deleteEntry(Entry entry) {
    return contents.remove(entry);
  }

  public void addEntry(Entry entry) {
    contents.add(entry);
  }

  protected ArrayList<Entry> getContents() { return contents; }
}
```

另外，我们还可以这样实现`Directory`：为文件和子目录创建不同的链表。如此一来，`numberOfFiles()`方法就不需要再用`instanceof`运算符，所以更为简洁，不过，我们就无法轻易按日期或名称对文件和目录进行排序。

8.10 设计并实现一个散列表，使用链接（即链表）处理碰撞冲突。（第 66 页）

解法

假设我们要实现类似`Hash<K, V>`的散列表。即，该散列表将类型`K`的对象映射为类型`V`的对象。

首先，我们或许会想到数据结构应该大致如下：

```
public class Hash<K, V> {
   LinkedList<V>[] items;
   public void put(K key, V value) { ... }
   public V get(K key) { ... }
}
```

注意，items是个链表的数组，其中items[i]是个链表，包含所有键映射成索引i的对象（也即在i处碰撞冲突的所有对象）。

这么做看似可行，不过要下定论，还得更深入一些考虑碰撞冲突的情况。

假设我们有个非常简单、使用字符串长度的散列函数。

```
public int hashCodeOfKey(K key) {
   return key.toString().length() % items.length;
}
```

键jim和bob都会对应到数组的同一索引，尽管这两个键并不一样。我们必须搜索整个链表，找出这些键对应的真正对象。但是该怎么办呢？我们在链表里存储的只有值，并不包括原先的键。

这就是要把值和原先的键一并存储起来的原因。

一种做法是引入一个Cell对象，存储键值对。在这种实现中，链表元素的类型为Cell。

下面是该实现的代码。

```
public class Hash<K, V> {
   private final int MAX_SIZE = 10;
   LinkedList<Cell<K, V>>[] items;

   public Hash() {
     items = (LinkedList<Cell<K, V>>[]) new LinkedList[MAX_SIZE];
   }

   /* 非常非常粗陋的散列 */
   public int hashCodeOfKey(K key) {
     return key.toString().length() % items.length;
   }

   public void put(K key, V value) {
     int x = hashCodeOfKey(key);
     if (items[x] == null) {
        items[x] = new LinkedList<Cell<K, V>>();
     }

     LinkedList<Cell<K, V>> collided = items[x];

     /* 查找有着相同键的项目，若找到则替换掉 */
     for (Cell<K, V> c : collided) {
        if (c.equivalent(key)) {
          collided.remove(c);
          break;
        }
     }

     Cell<K, V> cell = new Cell<K, V>(key, value);
     collided.add(cell);
   }

   public V get(K key) {
     int x = hashCodeOfKey(key);
     if (items[x] == null) {
```

```
            return null;
        }
        LinkedList<Cell<K, V>> collided = items[x];
        for (Cell<K, V> c : collided) {
            if (c.equivalent(key)) {
              return c.getValue();
            }
        }

        return null;
    }
}
```

Cell类存储有一对数据值和键。这样一来，我们就可以搜索整个链表（因碰撞冲突而建，但键不一样），找到对应该键值的对象。

```
public class Cell<K, V> {
   private K key;
   private V value;
   public Cell(K k, V v) {
     key = k;
     value = v;
   }

   public boolean equivalent(Cell<K, V> c) {
     return equivalent(c.getKey());
   }

   public boolean equivalent(K k) {
     return key.equals(k);
   }

   public K getKey() { return key; }
   public V getValue() { return value; }
}
```

实现散列表的另一种常见做法是使用二叉查找树作为底层数据结构。检索元素的时间复杂度不再是$O(1)$（不过，从技术上来说，复杂度不会是$O(1)$，因为可能有很多碰撞冲突），但是这种做法不需要创建一个无谓的大数组，用以存储项目。

9.9 递归和动态规划

9.1 有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上 1 阶、2 阶或 3 阶。实现一个方法，计算小孩有多少种上楼梯的方式。（第 68 页）

解法

我们可以采用自上而下的方式来解决这个问题。小孩上楼梯的最后一步，也就是抵达第n阶的那一步，可能走1阶、2阶或3阶。也就是说，最后一步可能是从第$n-1$阶往上走1阶、从第$n-2$

阶往上走2阶，或从第n–3阶往上走3阶。因此，抵达最后一阶的走法，其实就是抵达这最后三阶的方式的总和。

下面是该算法的简单实现。

```
public int countWays(int n) {
   if (n < 0) {
     return 0;
   } else if (n == 0) {
     return 1;
   } else {
     return countWays(n - 1) + countWays(n - 2) +
            countWays(n - 3);
   }
}
```

跟斐波那契数列问题一样，这个算法的运行时间呈指数级增长（准确地说是$O(3^N)$），因为每次调用都会分支出三次调用。这就意味着，对同一数值，countWays会调用很多次，而这显然没有必要。我们可以利用动态规划加以修正。

```
public static int countWaysDP(int n, int[] map) {
   if (n < 0) {
     return 0;
   } else if (n == 0) {
     return 1;
   } else if (map[n] > -1) {
     return map[n];
   } else {
     map[n] = countWaysDP(n - 1, map) +
              countWaysDP(n - 2, map) +
              countWaysDP(n - 3, map);
     return map[n];
   }
}
```

无论是否使用动态规划，注意上楼梯的方式总数很快就会突破整数（int型）的上限而溢出。当$n = 37$时，结果就会溢出。使用long可以撑久一点，但也不能从根本上解决问题。

9.2 设想有个机器人坐在$X \times Y$网格的左上角，只能向右、向下移动。机器人从(0,0)到(X, Y)有多少种走法？

进阶

假设有些点为“禁区”，机器人不能踏足。设计一种算法，找出一条路径，让机器人从左上角移动到右下角。**（第 68 页）**

解法

我们需要数一数机器人向右X步、向下Y步，总共可以走出多少种路径。这条路径总共有$X+Y$步。

为了走出一条路径，我们实质上要从$X+Y$步里，选出X步为向右移动。因此，可能路径的总数就是从$X+Y$项中选出X项的方法总数。具体可以用下面的二项式（又称“n选r”）表示：

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

对这个问题来说，该算式变成：

$$\binom{X+Y}{X} = \frac{(X+Y)!}{X!Y!}$$

就算不知道二项式，你也可以自行推导出解法。

我们可以将每条路径看作一个长度为$X+Y$的字符串，由X个R和Y个D组成。$X+Y$个不同的字符可以组成$(X+Y)!$个字符串。不过，在这个问题中，有X个字符为R，Y个字符为D。R有$X!$种排列组合，这些组合全都一样，对D的情况也类似，因此必须将结果除以$X!$和$Y!$。最后可得到跟前面相同的算式：

$$\frac{(X+Y)!}{X!Y!}$$

进阶：找到一条避开禁区点的路径

如果把网格画出来，你会发现移动到位置(X,Y)的唯一方式，就是先移动到它的相邻点：$(X-1,Y)$或$(X,Y-1)$。因此，我们需要找到一条移至$(X-1,Y)$或$(X,Y-1)$的路径。

怎么才能找出前往这些位置的路径呢？要找出前往$(X-1,Y)$或$(X,Y-1)$的路径，我们需要先移至其中一个相邻点。因此，要找到一条路径移动到$(X-1,Y)$的相邻点，坐标为$(X-2,Y)$和$(X-1,Y-1)$，或$(X,Y-1)$的相邻点，坐标为$(X-1,Y-1)$和$(X,Y-2)$。注意，坐标$(X-1,Y-1)$一共出现了两次；我们稍候再作讨论。

因此，要找到一条从原点出发的路径，我们只需像上面那样从终点往回走。从最后一点开始，试着找出一条到其相邻点的路径。下面是该算法的递归实现代码。

```
public boolean getPath(int x, int y, ArrayList<Point> path) {
   Point p = new Point(x, y);
   path.add(p);
   if (x == 0 && y == 0) {
     return true; // 找到一条路径
   }
   boolean success = false;
   if (x >= 1 && isFree(x - 1, y)) { // 试着向左
     success = getPath(x - 1, y, path); // 可行！向左走
   }
   if (!success && y >= 1 && isFree(x, y - 1)) { // 试着向上
     success = getPath(x, y - 1, path); // 可行！向上走
   }
   if (!success) {
     path.add(p); // 错了！最好不要再走这里
   }
   return success;
}
```

之前我们提到了重复路径的问题。要找到一条前往(X,Y)的路径，就要找出到它的相邻点$(X-1,Y)$和$(X,Y-1)$的路径。当然，若其中一个方格禁止通行，我们就要绕着走。接着，再看这两个点的相邻点：$(X-2,Y)$、$(X-1,Y-1)$、$(X-1,Y-1)$和$(X,Y-2)$。其中，$(X-1,Y-1)$出现了两次，也意味着我们做了一次无用功。理想情况下，我们应该记下先前访问过$(X-1,Y-1)$，以免浪费宝贵的时间。

下面就是运用了动态规划的算法。

```
public boolean getPath(int x, int y, ArrayList<Point> path,
                       Hashtable<Point, Boolean> cache) {
  Point p = new Point(x, y);
  if (cache.containsKey(p)) { // 已访问过这个点
    return cache.get(p);
  }
  path.add(p);
  if (x == 0 && y == 0) {
    return true; // 找到一条路径
  }
  boolean success = false;
  if (x >= 1 && isFree(x - 1, y)) { // 试着向左
    success = getPath(x - 1, y, path, cache); // 可行！向左走
  }
  if (!success && y >= 1 && isFree(x, y - 1)) { // 试着向上
    success = getPath(x, y - 1, path, cache); // 可行！向上走
  }
  if (!success) {
    path.add(p); // 错了！最好不要再走这里
  }
  cache.put(p, success); // 缓存结果
  return success;
}
```

只要稍作修改，就能大幅提升程序的执行速度。

9.3 在数组A[0...n-1]中，有所谓的魔术索引，满足条件A[i] = i。给定一个有序整数数组，元素值各不相同，编写一个方法，在数组A中找出一个魔术索引，若存在的话。

进阶

如果数组元素有重复值，又该如何处理？（第 68 页）

解法

看到这个问题，第一反应可能是选择蛮力法，这也没什么好羞愧的。我们可以直接迭代访问整个数组，找出符合条件的元素。

```
public static int magicSlow(int[] array) {
  for (int i = 0; i < array.length; i++) {
    if (array[i] == i) {
      return i;
    }
  }
  return -1;
}
```

不过，既然给定数组是有序的，我们理应充分利用这个条件。

你可能会发现这个问题与经典的二分查找问题非常相似。充分运用模式匹配法，就能找出适当的算法，我们又该怎么运用二分查找法呢？

在二分查找中，要找出元素*k*，我们会先拿它跟数组中间的元素*x*比较，确定*k*位于*x*的左边还是右边。

以此为基础，是否通过检查中间元素就能确定魔术索引的位置？下面来看一个样例数组：

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

看到中间元素`A[5] = 3`，我们可以断定魔术索引一定在数组右侧，因为`A[mid] < mid`。

为何魔术索引不会在数组左侧呢？注意，从元素*i*移至*i*–1时，此索引对应的值至少要减1，也可能更多（因为数组是有序的，且所有元素各不相同）。因此，如果中间元素就已经太小而不是魔术索引的话，那么往左侧移动时，索引减*k*，值至少也减*k*，所有余下的元素也会太小。

继续运用这个递归算法，就会写出与二分查找非常相似的代码。

```
public static int magicFast(int[] array, int start, int end) {
   if (end < start || start < 0 || end >= array.length) {
     return -1;
   }
   int mid = (start + end) / 2;
   if (array[mid] == mid) {
     return mid;
   } else if (array[mid] > mid){
     return magicFast(array, start, mid - 1);
   } else {
     return magicFast(array, mid + 1, end);
   }
}

public static int magicFast(int[] array) {
   return magicFast(array, 0, array.length - 1);
}
```

进阶：如果数组元素有重复值又该如何处理？

如果数组元素有重复值，前面的算法就会失效。以下面的数组为例：

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

看到`A[mid] < mid`时，我们无法断定魔术索引位于数组哪一边。它可能在数组右侧，跟前面一样。或者，也可能在左侧（在本例中的确在左侧）。

它有没有可能在左侧的任意位置？未必。由`A[5] = 3`可知，`A[4]`不可能是魔术索引。`A[4]`必须等于4，其索引才能成为魔术索引，但数组是有序的，故`A[4]`必定小于`A[5]`。

事实上，看到`A[5] = 3`时，按照前面的做法，我们需要递归搜索右半部分。不过，若搜索左半部分，我们可以跳过一些元素，只递归搜索`A[0]`到`A[3]`的元素。`A[3]`是第一个可能成为魔术索引的元素。

综上，我们得到一般模式：先比较`midIndex`和`midValue`是否相同。然后，若两者不同，则按如下方式递归搜索左半部分和右半部分。

- 左半部分：搜索索引从`start`到`Math.min(midIndex - 1, midValue)`的元素。
- 右半部分：搜索索引从`Math.max(midIndex + 1, midValue)`到`end`的元素。

下面是该算法的实现代码。

```
public static int magicFast(int[] array, int start, int end) {
   if (end < start || start < 0 || end >= array.length) {
     return -1;
   }
   int midIndex = (start + end) / 2;
   int midValue = array[midIndex];
   if (midValue == midIndex) {
     return midIndex;
   }

   /* 搜索左半部分 */
   int leftIndex = Math.min(midIndex - 1, midValue);
   int left = magicFast(array, start, leftIndex);
   if (left >= 0) {
     return left;
   }

   /* 搜索右半部分 */
   int rightIndex = Math.max(midIndex + 1, midValue);
   int right = magicFast(array, rightIndex, end);

   return right;
}

public static int magicFast(int[] array) {
   return magicFast(array, 0, array.length - 1);
}
```

注意，在上面的代码中，如果数组元素各不相同，这个方法的执行动作与第一个解法几近相同。

9.4　编写一个方法，返回某集合的所有子集。（第 68 页）

解法

着手解决这个问题之前，我们先要对时间和空间复杂度有个合理的评估。一个集合会有多少子集？我们可以这么计算，生成一个子集时，每个元素都可以“选择”在或不在这个子集中。也就是说，第一个元素有两个选择：它要么在集合中，要么不在集合中。同样，第二个元素也有两个选择，依此类推，2相乘n次，`{2 * 2 * … }`等于2^n个子集。因此，在时间或空间复杂度上，我们不可能做得比$O(2^n)$更好。

集合$\{a_1, a_2, \cdots, a_n\}$的所有子集组成的集合也称为幂集(powerset),用符号表示为:$P(\{a_1, a_2, \cdots, a_n\})$或$P(n)$。

解法 1:递归

此题非常适合采用简单构造法。假设我们正尝试找出集合$S = \{a_1, a_2, ..., a_n\}$的所有子集,可从终止条件开始。

- 终止条件:$n = 0$

空集合只有一个子集:{}。

- 情况:$n = 1$

集合$\{a_1\}$有两个子集:{}、$\{a_1\}$。

- 情况:$n = 2$

集合$\{a_1, a_2\}$有四个子集:{}、$\{a_1\}$、$\{a_2\}$、$\{a_1, a_2\}$。

- 情况:$n = 3$

至此,事情开始变得有点意思了。我们想找出一种方法,可以根据之前的答案产生$n = 3$时的答案。

$n = 3$和$n = 2$的两个答案之间有何不同?下面让我们更细致地分析两者差异:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$

$$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

两者之间的不同之处在于,所有含有a_3的子集,$P(2)$都没有。

$$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

那么,我们该如何利用$P(2)$构造$P(3)$?很简单,只需复制$P(2)$里的子集,并在这些子集中添加a_3:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$

$$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

两者合并在一起,即可产生$P(3)$。

- 情况:$n > 0$

只要将上述步骤稍作一般化处理,就能产生一般情况的$P(n)$:先计算$P(n-1)$,复制一份结果,然后在每个复制后的集合中加入a_n。

下面是该算法的实现代码:

```
ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set,
                                         int index) {
  ArrayList<ArrayList<Integer>> allsubsets;
  if (set.size() == index) { // 终止条件,加入空集合
    allsubsets = new ArrayList<ArrayList<Integer>>();
    allsubsets.add(new ArrayList<Integer>()); // 空集合
  } else {
    allsubsets = getSubsets(set, index + 1);
    int item = set.get(index);
    ArrayList<ArrayList<Integer>> moresubsets =
```

```
            new ArrayList<ArrayList<Integer>>();
        for (ArrayList<Integer> subset : allsubsets) {
            ArrayList<Integer> newsubset = new ArrayList<Integer>();
            newsubset.addAll(subset);
            newsubset.add(item);
            moresubsets.add(newsubset);
        }
        allsubsets.addAll(moresubsets);
    }
    return allsubsets;
}
```

这个解法的时间和空间复杂度为$O(2^n)$，已经是最优解。非要锦上添花的话，我们还可以迭代方式实现这个算法。

解法 2：组合数学（Combinatorics）

尽管上面的解法没什么地方不对，不过还是可以另觅他法，解决这个问题。

回想一下，在构造一个集合时，每个元素有两个选择：（1）该元素在这个集合中（“yes”状态），或者（2）该元素不在这个集合中（“no”状态）。这就意味着每个子集都是一串yes和no，比如“yes, yes, no, no, yes, no”。

由此，总共可能会有2^n个子集。怎样才能迭代遍历所有元素的所有“yes” / “no”序列？如果将每个“yes”视作1，每个“no”视作0，那么，每个子集就可以表示为一个二进制串。

接着，构造所有子集就等同于构造所有的二进制数（也即所有整数）。我们会迭代访问1到2^n的所有数字，再将这些数字的二进制表示转换成集合。小事一桩！

```
ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
    ArrayList<ArrayList<Integer>> allsubsets =
        new ArrayList<ArrayList<Integer>>();
    int max = 1 << set.size(); /* 计算2^n */
    for (int k = 0; k < max; k++) {
        ArrayList<Integer> subset = convertIntToSet(k, set);
        allsubsets.add(subset);
    }
    return allsubsets;
}

ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
    ArrayList<Integer> subset = new ArrayList<Integer>();
    int index = 0;
    for (int k = x; k > 0; k >>= 1) {
        if ((k & 1) == 1) {
            subset.add(set.get(index));
        }
        index++;
    }
    return subset;
}
```

相比前一种解法，这种解法不存在实质的差异，并无上下之分。

9.5 编写一个方法，确定某字符串的所有排列组合。（第 68 页）

解法

跟许多递归问题一样，简单构造法非常管用。假设有个字符串S，以字符序列$a_1a_2\cdots a_n$表示。

- 终止条件：$n=1$

$S=a_1$，只有一种排列组合，即字符串a_1。

- 情况：$n=2$

$S=a_1a_2$，有两种排列组合a_1a_2和a_2a_1。

- 情况：$n=3$

至此，情况变得越来越有意思。根据a_1a_2的排列组合，如何产生$a_1a_2a_3$的所有排列组合呢？也就是说，给定

$$a_1a_2,\ a_2a_1$$

我们需要产生：

$$a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1$$

这两个字符序列的区别在于前者不含a_3，而后者包含a_3。那么，怎样才能根据$f(2)$生成$f(3)$呢？很简单，将a_3塞进$f(2)$里所有字符串的任意可能位置即可。

- 情况：$n>0$

对于一般情况，我们只需重复这个步骤。既然已求得$f(n-1)$的解，接着只要将a_n插入这些字符串的任意位置。

具体代码如下。

```
public static ArrayList<String> getPerms(String str) {
   if (str == null) {
     return null;
   }
   ArrayList<String> permutations = new ArrayList<String>();
   if (str.length() == 0) { // 终止条件
     permutations.add(“”);
     return permutations;
   }

   char first = str.charAt(0); // 取得第一个字符
   String remainder = str.substring(1); // 移除第一个字符
   ArrayList<String> words = getPerms(remainder);
   for (String word : words) {
     for (int j = 0; j <= word.length(); j++) {
        String s = insertCharAt(word, first, j);
        permutations.add(s);
     }
   }
   return permutations;
}

public static String insertCharAt(String word, char c, int i) {
   String start = word.substring(0, i);
```

```
    String end = word.substring(i);
    return start + c + end;
}
```

由于将会有$n!$种排列组合，这种解法的时间复杂度为$O(n!)$，已经是最优解。

9.6 实现一种算法，打印n对括号的全部有效组合（即左右括号正确配对）。（第 68 页）

解法

看到此题，我们的第一反应可能是运用递归法，将一对括号加进$f(n-1)$的解答，从而得到$f(n)$的解答。从直觉上看，这个方法不错。

下面来看看$n = 3$时的答案：

```
(()())    ((()))    ()(())    (())()    ()()()
```

如何以$n = 2$时的答案为基础构建上面的结果呢？

```
(())    ()()
```

我们可以在字符串最前面以及原有的每对括号里面插入一对括号。至于插入其他任意位置，比如字符串的末尾，都会跟之前的情况重复。

综上，可得到以下结果：

```
(()) -> (()()) /* 在第1个左括号之后插入一对括号 */
     -> ((())) /* 在第2个左括号之后插入一对括号 */
     -> ()(()) /* 在字符串开头插入一对括号 */
()() -> (())() /* 在第1个左括号之后插入一对括号 */
     -> ()(()) /* 在第2个左括号之后插入一对括号 */
     -> ()()() /* 在字符串开头插入一对括号 */
```

且慢，上面有重复的括号对组合，`()(())`出现了两次。

如果准备采用这种做法，那么，将字符串放进结果列表之前，必须先检查有无重复。

```
public static Set<String> generateParens(int remaining) {
   Set<String> set = new HashSet<String>();
   if (remaining == 0) {
     set.add(“”);
   } else {
     Set<String> prev = generateParens(remaining - 1);
     for (String str : prev) {
        for (int i = 0; i < str.length(); i++) {
          if (str.charAt(i) == ‘(‘) {
             String s = insertInside(str, i);
             /* 若s不在set中，则将s插入set中。注意，
              * 插入元素之前，HashSet会自动检查有无重复，
              * 因此没必要专门检查元素是否重复 */
             set.add(s);
          }
        }
        if (!set.contains(“()” + str)) {
           set.add(“()” + str);
        }
     }
```

```
    }
    return set;
}

public String insertInside(String str, int leftIndex) {
    String left = str.substring(0, leftIndex + 1);
    String right = str.substring(leftIndex + 1, str.length());
    return left + “()” + right;
}
```

这种做法可行，但效率不太高，在排查重复字符串上浪费了大量时间。

另一种解法是从头开始构造字符串，从而避免出现重复字符串。在这个解法中，逐一加入左括号和右括号，只要字符串仍然有效（合乎题意）。

每次递归调用，都会有个索引值向字符串的某个字符。我们需要选择左括号或右括号，那么，何时可以用左括号，何时可以用右括号呢？

(1) *左括号*：只要左括号还没有用完，就可以插入左括号。

(2) *右括号*：只要不造成语法错误，就可以插入右括号。何时会出现语法错误？如果右括号比左括号还多，就会出现语法错误。

因此，我们只需记录允许插入的左右括号数目。如果还有左括号可用，就插入一个左括号然后递归。如果右括号比左括号还多（也就是使用中的左括号比右括号还多），就插入一个右括号然后递归。

```
public void addParen(ArrayList<String> list, int leftRem,
                     int rightRem, char[] str, int count) {
    if (leftRem < 0 || rightRem < leftRem) return; // 无效状态

    if (leftRem == 0 && rightRem == 0) { /* 没有括号可用了 */
      String s = String.copyValueOf(str);
      list.add(s);
    } else {
      /* 若还有左括号可用，则加入一个左括号 */
      if (leftRem > 0) {
         str[count] = ‘(’;
         addParen(list, leftRem - 1, rightRem, str, count + 1);
      }

      /* 若字符串是有效的，则加入右括号 */
      if (rightRem > leftRem) {
         str[count] = ‘)’;
         addParen(list, leftRem, rightRem - 1, str, count + 1);
      }
    }
}

public ArrayList<String> generateParens(int count) {
    char[] str = new char[count*2];
    ArrayList<String> list = new ArrayList<String>();
    addParen(list, count, count, str, 0);
    return list;
}
```

因为我们是在字符串的每一个索引对应位置插入左括号和右括号，而且绝不会重复索引，所以，可以保证每个字符串都是独一无二的。

9.7　编写函数，实现许多图片编辑软件都支持的“填充颜色”功能。给定一个屏幕（以二维数组表示，元素为颜色值）、一个点和一个新的颜色值，将新颜色值填入这个点的周围区域，直到原来的颜色值全都改变。**（第 69 页）**

解法

首先，想象一下这个方法是怎么回事。假设要对一个像素（比如绿色）调用`paintFill`（也即点击图片编辑软件的填充颜色），我们希望颜色向四周“渗出”。我们会对周围的像素逐一调用`paintFill`，向外扩张，一旦碰到非绿色的像素就停止填充。

我们可以递归方式实现这个算法：

```
enum Color {
   Black, White, Red, Yellow, Green
}

boolean paintFill(Color[][] screen, int x, int y, Color ocolor,
                    Color ncolor) {
   if (x < 0 || x >= screen[0].length ||
      y < 0 || y >= screen.length) {
     return false;
   }
   if (screen[y][x] == ocolor) {
     screen[y][x] = ncolor;
     paintFill(screen, x - 1, y, ocolor, ncolor); // 左
     paintFill(screen, x + 1, y, ocolor, ncolor); // 右
     paintFill(screen, x, y - 1, ocolor, ncolor); // 上
     paintFill(screen, x, y + 1, ocolor, ncolor); // 下
   }
   return true;
}

boolean paintFill(Color[][] screen, int x, int y, Color ncolor) {
   if (screen[y][x] == ncolor) return false;
   return paintFill(screen, x, y, screen[y][x], ncolor);
}
```

注意`screen`[*y*][*x*]中*x*和*y*的顺序，碰到图像问题时切记这一点。因为x表示水平轴（也即自左向右），实际上对应于列数而非行数。*y*的值等于行数。在面试以及平时写代码时，这个地方也很容犯错。

9.8　给定数量不限的硬币，币值为 25 分、10 分、5 分和 1 分，编写代码计算*n*分有几种表示法。**（第 69 页）**

解法

这是个递归问题，我们要找出如何利用较早的答案（也就是子问题的答案）计算出`makeChange(n)`。

假设$n = 100$，我们想要算出100分有几种换零方式。这个问题与其子问题之间有何关系呢？

我们知道100分换零后会包含0、1、2、3或4个25分硬币（quarter），因此：

```
makeChange(100) =
    makeChange(100，使用0个25分硬币) +
    makeChange(100，使用1个25分硬币) +
    makeChange(100，使用2个25分硬币) +
    makeChange(100，使用3个25分硬币) +
    makeChange(100，使用4个25分硬币)
```

仔细观察一番，可以看出其中有些问题简化掉了。举个例子，`makeChange(100，使用1个25分硬币)`与`makeChange(75，使用0个25分硬币)`等价。这是因为，如果给100分换零时只准用1个25分硬币，那么，我们就只能选择给余下的75分换零。

同样的逻辑也适用于`makeChange(100，使用2个25分硬币)`、`makeChange(100，使用3个25分硬币)`和`makeChange(100，使用4个25分硬币)`。综上，前面的算式可简化为：

```
makeChange(100) =
    makeChange(100，使用0个25分硬币) +
    makeChange(75，使用0个25分硬币) +
    makeChange(50，使用0个25分硬币) +
    makeChange(25，使用0个25分硬币) +
    1
```

注意最后一行，`makeChange(100，使用4个25分硬币)`等于1。我们把这叫作“完全简化”。

接下来呢？我们已经用完了25分硬币，现在可以开始使用下一个币值最大的硬币：10分硬币（dime）。

前面使用25分硬币的做法同样可以套用在10分硬币上，但需要套用在上面算式五部分中的四个部分，且每一部分都要套用。第一部分的套用结果如下：

```
makeChange(100，使用0个25分硬币) =
    makeChange(100，使用0个25分硬币、0个10分硬币) +
    makeChange(100，使用0个25分硬币、1个10分硬币) +
    makeChange(100，使用0个25分硬币、2个10分硬币) +
    ...
    makeChange(100，使用0个25分硬币、10个10分硬币)

makeChange(75，使用0个25分硬币) =
    makeChange(75，使用0个25分硬币、0个10分硬币) +
    makeChange(75，使用0个25分硬币、1个10分硬币) +
    makeChange(75，使用0个25分硬币、2个10分硬币) +
    ...
    makeChange(75，使用0个25分硬币、7个10分硬币)

makeChange(50，使用0个25分硬币) =
    makeChange(50，使用0个25分硬币、0个10分硬币) +
    makeChange(50，使用0个25分硬币、1个10分硬币) +
    makeChange(50，使用0个25分硬币、2个10分硬币) +
    ...
    makeChange(50，使用0个25分硬币、5个10分硬币)
```

```
makeChange(25，使用0个25分硬币) =
    makeChange(25，使用0个25分硬币、0个10分硬币) +
    makeChange(25，使用0个25分硬币、1个10分硬币) +
    makeChange(25，使用0个25分硬币、2个10分硬币)
```

开始使用5分镍币（nickel）时，上面算式的每一部分都要逐一展开，最终会得到一个树状递归结构，其中每次调用都会展开为4个或更多调用。

递归的终止条件就是完全简化的算式。举个例子，makeChange(50，使用0个25分硬币、5个10分硬币)会被完全简化为1，因为5个10分硬币就等于50分。

由上述说明可导出类似如下的递归算法：

```
public int makeChange(int n, int denom) {
   int next_denom = 0;
   switch (denom) {
   case 25:
     next_denom = 10;
     break;
   case 10:
     next_denom = 5;
     break;
   case 5:
     next_denom = 1;
     break;
   case 1:
     return 1;
   }

   int ways = 0;
   for (int i = 0; i * denom <= n; i++) {
     ways += makeChange(n - i * denom, next_denom);
   }
   return ways;
}

System.out.writeln(makeChange(100, 25));
```

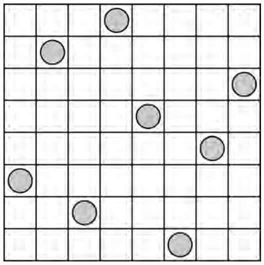
上面的算法只适用于美国币值，不过，只要稍加修改扩充，就能用于其他的币值组合。

9.9　设计一种算法，打印八皇后在 8×8 棋盘上的各种摆法，其中每个皇后都不同行、不同列，也不在对角线上。这里的“对角线”指的是所有的对角线，不只是平分整个棋盘的那两条对角线。（第 69 页）

解法

我们必须在8×8棋盘上排好8个皇后，每个皇后都位于不同行、不同列，且不在同一对角线上。由此可知，每一行、每一列以及对角线只能使用一次。

想象一下最后放到棋盘上的那个皇后，这里假设是在第8行。（这么假设没有问题，因为这些皇后怎么摆放都没关系。）这个皇后要摆在第8行的哪一格呢？一共有8种选择，每一列代表一种可能。



“摆好”八皇后的棋盘，其中一种摆法

因此，欲知八皇后在8×8棋盘上的所有可能摆法，具体算法如下：

```
八皇后在8×8棋盘上的摆法 =
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 0) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 1) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 2) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 3) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 4) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 5) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 6) +
   八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 7)
```

接着，运用非常类似的方法计算其中的每一项：

```
八皇后在8×8棋盘上的摆法，且其中一个皇后位于(7, 3) =
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 0) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 1) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 2) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 4) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 5) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 6) +
   八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 7)
```

注意，我们不必考虑皇后位于格子(7, 3)和(6, 3)的组合情况，因为这与所有皇后不同行、不同列且不在对角线上的要求不符。

接下来，具体实现也就相当简单了。

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns,
                                           ArrayList<Integer[]> results) {
   if (row == GRID_SIZE) { // 找到有效的摆法
     results.add(columns.clone());
   } else {
     for (int col = 0; col < GRID_SIZE; col++) {
        if (checkValid(columns, row, col)) {
           columns[row] = col;  // 摆放皇后
           placeQueens(row + 1, columns, results);
        }
     }
   }
}
```

```
/* 检查(row1, column1)可否摆放皇后，做法是
 * 检查有无其他皇后位于同一列或对角线，不必
 * 检查是否在同一行上，因为调用placeQueen时，
 * 一次只会摆放一个皇后，由此可知，这一行是
 * 空的*/
boolean checkValid(Integer[] columns, int row1, int column1) {
   for (int row2 = 0; row2 < row1; row2++) {
     int column2 = columns[row2];
     /* 检查(row2, column2)是否会让(row1, column1)变成无效
      * 摆放位置 */

     /* 检查同一列有无其他皇后 */
     if (column1 == column2) {
        return false;
     }

     /* 检查对角线：若两列的距离等于
      * 两行的距离，就表示两个皇后在
      * 同一对角线上 */
     int columnDistance = Math.abs(column2 - column1);

     /* row1 > row2，不用取绝对值 */
     int rowDistance = row1 - row2;
     if (columnDistance == rowDistance) {
        return false;
     }
   }
   return true;
}
```

注意，每一行只能摆放一个皇后，因此不需要将棋盘储存为完整的8×8矩阵，只需一维数组，其中`columns[r] = c`表示有个皇后位于行r列c。

9.10　给你一堆*n*个箱子，箱子宽w_i、高h_i、深d_i。箱子不能翻转，将箱子堆起来时，下面箱子的宽度、高度和深度必须大于上面的箱子。实现一个方法，搭出最高的一堆箱子，箱堆的高度为每个箱子高度的总和。（第 69 页）

解法

要解决此题，我们需要找到不同子问题之间的关系。

假设我们有以下这些箱子：$b_1, b_2, \cdots, b_n$。能够堆出的最高箱堆的高度等于max(底部为b_1的最高箱堆, 底部为b_2的最高箱堆, ⋯, 底部为b_n的最高箱堆)。也就是说，只要试着用每个箱子作为箱堆底部并搭出可能的最高高度，就能找出箱堆的最高高度。

但是，该怎么找出以某个箱子为底的最高箱堆呢？具体做法与之前的完全相同。我们会试着在第二层以不同的箱子为底继续堆箱子，如此反复。

当然，我们只需尝试有效的箱子，也就是说，若b_5大于b_1，那就不必尝试这么堆箱子：$\{b_1, b_5, \cdots\}$，因为b_1不能放在b_5下面。

下面是该算法的递归实现代码。

```
public ArrayList<Box> createStackR(Box[] boxes, Box bottom) {
   int max_height = 0;
   ArrayList<Box> max_stack = null;
   for (int i = 0; i < boxes.length; i++) {
     if (boxes[i].canBeAbove(bottom)) {
        ArrayList<Box> new_stack = createStackR(boxes, boxes[i]);
        int new_height = stackHeight(new_stack);
        if (new_height > max_height) {
           max_stack = new_stack;
           max_height = new_height;
        }
     }
   }

   if (max_stack == null) {
     max_stack = new ArrayList<Box>();
   }
   if (bottom != null) {
     max_stack.add(0, bottom); // 插入箱堆底部
   }

   return max_stack;
}
```

上述代码的问题是效率太低，我们可能已经找出以b_4为底的最优解，但还是尝试找到类似$\{b_3, b_4, \cdots\}$的最佳解决方案。我们不必像之前那样从零开始构造这些答案，完全可以运用动态规划，缓存这些结果。

```
public ArrayList<Box> createStackDP(Box[] boxes, Box bottom,
     HashMap<Box, ArrayList<Box>> stack_map) {
   if (bottom != null && stack_map.containsKey(bottom)) {
     return stack_map.get(bottom);
   }

   int max_height = 0;
   ArrayList<Box> max_stack = null;
   for (int i = 0; i < boxes.length; i++) {
     if (boxes[i].canBeAbove(bottom)) {
        ArrayList<Box> new_stack =
           createStackDP(boxes, boxes[i], stack_map);
        int new_height = stackHeight(new_stack);
        if (new_height > max_height) {
           max_stack = new_stack;
           max_height = new_height;
        }
     }
   }

   if (max_stack == null) max_stack = new ArrayList<Box>();
   if (bottom != null) max_stack.add(0, bottom);
   stack_map.put(bottom, max_stack);

```

```
25    return (ArrayList<Box>)max_stack.clone();
26 }
```

你可能会问，第25行代码为什么非要转型max_stack.clone()，max_stack不就已经是正确的数据类型了吗？没错，但我们还是需要进行转型。

方法clone()来自Object类，其方法签名如下：

```
protected Object clone() { ... }
```

重写方法时，可以调整参数，但不得改动返回类型。因此，如果继承自Object的Foo类重写了clone()，它的clone()方法仍将返回Object实例。

这正是语句(ArrayList<Box>)max_stack.clone()的真正作用。这个类会重写clone()，但该方法仍然会返回Object，因此，我们必须转型返回值。

9.11　给定一个布尔表达式，由 0、1、&、|和^等符号组成，以及一个想要的布尔结果result，实现一个函数，算出有几种括号的放法可使该表达式得出result值。（第 69 页）

解法

跟其他递归问题一样，此题的关键在于找出问题与子问题之间的关系。

假设函数int f(expression, result)会返回所有值为return的有效表达式的数量。我们想要算出f(1^0|0|1, true)（也即，给表达式1^0|0|1加括号使其求值为true的所有方式）。每个加括号的表达式最外层肯定有一对括号。因此，我们可以这么做：

```
f(1^0|0|1, true) = f(1 ^ (0|0|1), true) +
                   f((1^0) | (0|1), true) +
                   f((1^0|0) | 1, true)
```

也就是说，我们可以迭代整个表达式，将每个运算符当作第一个要加括号的运算符。

现在，又该如何计算这些内层的表达式呢，比如f((1^0) | (0|1), true)？很简单，要让这个表达式的值为true，左半部分或右半部分必有其一为true。因此，这个表达式分解如下：

```
f((1^0) | (0|1), true) =   f(1^0, true)  * f(0|1, true) +
                           f(1^0, false) * f(0|1, true) +
                           f(1^0, true)  * f(0|1, false)
```

对每个布尔运算符，都可以进行类似的分解：

```
f(exp1 | exp2, true)   =   f(exp1, true)  * f(exp2, true) +
                           f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
f(exp1 & exp2, true)   =   f(exp1, true)  * f(exp2, true)
f(exp1 ^ exp2, true)   =   f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
```

对于false结果，我们也可以执行非常类似的操作：

```
f(exp1 | exp2, false)  =   f(exp1, false) * f(exp2, false)
f(exp1 & exp2, false)  =   f(exp1, false) * f(exp2, false) +
                           f(exp1, true)  * f(exp2, false) +
                           f(exp1, false) * f(exp2, true)
f(exp1 ^ exp2, false)  =   f(exp1, true)  * f(exp2, true) +
                           f(exp1, false) * f(exp2, false)
```

至此，要解决这个问题，只需反复套用这些递归关系即可。（注意：为了避免代码行不必要的回绕，以及确保代码的可读性，下面的代码使用了非常短的变量名。）

```
public int f(String exp, boolean result, int s, int e) {
   if (s == e) {
     if (exp.charAt(s) == '1' && result) {
        return 1;
     } else if (exp.charAt(s) == '0' && !result) {
        return 1;
     }
     return 0;
   }
   int c = 0;
   if (result) {
     for (int i = s + 1; i <= e; i += 2) {
        char op = exp.charAt(i);
        if (op == '&') {
           c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
        } else if (op == '|') {
           c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
           c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
           c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
        } else if (op == '^') {
           c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
           c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
        }
     }
   } else {
     for (int i = s + 1; i <= e; i += 2) {
        char op = exp.charAt(i);
        if (op == '&') {
           c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
           c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
           c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
        } else if (op == '|') {
           c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
        } else if (op == '^') {
           c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
           c += f(exp, false, s, i - 1) * f(exp, false, i + 1,e);
        }
     }
   }
   return c;
}
```

虽然这么做可行，但不是很有效，对于同一个exp的值，它会重算f(exp)很多次。

要解决这个问题，我们可以运用动态规划，缓存不同表达式的结果。注意，我们需要根据expression和result进行缓存。

```
public int f(String exp, boolean result, int s, int e,
             HashMap<String, Integer> q) {
   String key = "" + result + s + e;
   if (q.containsKey(key)) {
```

```
    return q.get(key);
  }

  if (s == e) {
    if (exp.charAt(s) == ‘1’ && result == true) {
       return 1;
    } else if (exp.charAt(s) == ‘0’ && result == false) {
       return 1;
    }
    return 0;
  }
  int c = 0;
  if (result) {
    for (int i = s + 1; i <= e; i += 2) {
       char op = exp.charAt(i);
       if (op == ‘&’) {
         c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
       } else if (op == ‘|’) {
         c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
         c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
         c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
       } else if (op == ‘^’) {
         c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
         c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
       }
    }
  } else {
    for (int i = s + 1; i <= e; i += 2) {
       char op = exp.charAt(i);
       if (op == ‘&’) {
         c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
         c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
         c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
       } else if (op == ‘|’) {
         c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
       } else if (op == ‘^’) {
         c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
         c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
       }
    }
  }
  q.put(key, c);
  return c;
}
```

运用动态规划后，虽然该算法已得到很好的优化，但还不够最优。要是知道一个表达式有多少种括号的放法，我们完全可以借由`total(exp) - f(exp = true)`来算出`f(exp = false)`。

对于一个表达式有几种括号的放法，的确有个公式解，只是你可能不知道罢了。这个解可由卡塔兰数导出，其中n为运算符的数目：

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

经过这次调整，实现代码类似如下：

```
public int f(String exp, boolean result, int s, int e,
        HashMap<String, Integer> q) {
   String key = “” + s + e;
   int c = 0;
   if (!q.containsKey(key)) {
     if (s == e) {
        if (exp.charAt(s) == ‘1’) c = 1;
        else c = 0;
     }

     for (int i = s + 1; i <= e; i += 2) {
        char op = exp.charAt(i);
        if (op == ‘&’) {
           c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
        } else if (op == ‘|’) {
           int left_ops = (i-1-s)/2; // 括号在左边
           int right_ops = (e - i - 1) / 2;  // 括号在右边
           int total_ways = total(left_ops) * total(right_ops);
           int total_false = f(exp,false,s,i-1,q) *
                             f(exp,false,i+1,e,q);
           c += total_ways - total_false;
        } else if (op == ‘^’) {
           c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
           c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
        }
     }
     q.put(key, c);
   } else {
     c = q.get(key);
   }
   if (result) {
     return c;
   } else {
     int num_ops = (e - s) / 2;
     return total(num_ops) - c;
   }
}
```

9.10　扩展性与存储限制

10.1　假设你正在搭建某种服务，有多达 1000 个客户端软件会调用该服务，取得每天盘后股票价格信息（开盘价、收盘价、最高价与最低价）。假设你手里已有这些数据，存储格式可自行定义。你会如何设计这套面向客户端的服务，向客户端软件提供信息？你将负责该服务的研发、部署、持续监控和维护。描述你想到的各种实现方案，以及为何推荐采用你的方案。该服务的实现技术可任选，此外，可以选用任何机制向客户端分发信息。（第 72 页）

解法

从此题描述来看，我们要关注的是如何真正地将信息分发给客户端。在此假定有一些脚本可

以神奇地把信息收集起来。

首先，让我们想一想合乎要求的方案应该具备哪几方面。

- **客户端软件易用性**：我们希望这套服务对客户端实现起来又容易又好用。
- **让我们自己实现起来也轻松**：这套服务应该是越容易实现越好，不该自讨苦吃，把不必要的工作强加到自己头上。之所以要考虑这点，不仅是因为研发成本，还有维护成本。
- **灵活应对未来需求**：此题的问法是“在现实世界中你会怎么做”，因此，我们应该从解决实际问题的角度来思考。理想情况下，我们不想受到实现的过多限制，以致无法灵活应对条件或需求变更。
- **扩展性和效率**：关注实现方案的效率，才不会让服务负担过重。

有了这些注意事项，我们就可以考虑各种方案了。

方案 1

一种选择是，将数据直接保存在纯文本文件中，让客户端通过某种FTP服务器下载。从某种角度来说，这么做容易维护，因为可以自如地查看和备份这些文件，但需要更复杂的文件解析才能实现各种查询。此外，若这些文件有新增数据，可能会打乱客户端的解析机制。

方案 2

我们可以使用标准的SQL数据库，让客户端直接接入。这么做有如下优点。

- 如需支持新功能，这种做法提供了一种让客户端查询和处理数据的简单方式。例如，我们可以轻松、高效地执行这类查询：返回开盘价高于*N*且收盘价低于*M*的所有股票。
- 利用标准的数据库功能就能提供数据回滚、数据备份和各种安全保障。我们不必做无谓的重复性劳动，因此实现起来非常轻松。
- 客户端可以很容易地整合现有应用。在各种软件开发环境中，SQL整合是标准功能。

那么，使用SQL数据库有哪些缺点呢？

- 相比我们真正需要的，它所造成的负担过重。为了提供一些信息，我们并不一定需要SQL后端的所有复杂功能。
- 对用户来说，数据库基本不可读，因此需要多一层实现，以查看和维护数据。而这会增加实现成本。
- 安全性：尽管SQL数据库提供了非常明确的安全等级，我们还是要谨慎行事，不让客户端存取它们不该访问的数据。此外，即使客户端不会有“恶意”的动作，它们也可能执行昂贵和低效的查询，而我们的服务器将会承担这些开销。

列出这些缺点并不表示我们不该使用SQL。相反，列出它们是为了让我们对这些缺点心知肚明。

方案 3

就分发信息而言，XML也是一种不错的选择。采用XML时，数据有固定的格式和大小：`company name`（公司名）、`open`（开盘价）、`high`（最高价）、`low`（最低价）、`closingPrice`（收盘价），下面是一个XML格式的数据样例：

```
<root>
  <date value="2008-10-12">
    <company name="foo">
      <open>126.23</open>
      <high>130.27</high>
      <low>122.83</low>
      <closingPrice>127.30</closingPrice>
    </company>
    <company name="bar">
       <open>52.73</open>
       <high>60.27</high>
       <low>50.29</low>
       <closingPrice>54.91</closingPrice>
    </company>
  </date>
  <date value="2008-10-11"> . . . </date>
</root>
```

这种做法有如下优点。

- 容易分发，也容易为机器和人类识别。这也是XML成为分享和分发数据的标准数据模型的原因之一。
- 大多数语言都有执行XML解析的库，因此客户端实现起来也很容易。
- 在XML文件中增加新结点就可以添加新数据。这不会打乱客户端解析器（只要以正确的方式实现解析器）。
- 数据以XML文件格式存储，因此我们可以利用现有工具备份数据，不必自己重新做一套。

这么做可能有以下缺点。

- 这种做法会向客户端发送所有信息，即使他们只需要其中一部分。这么做效率很低。
- 进行数据查询时，必须解析整个文件。

无论采用哪种数据存储方案，我们都可以提供Web服务（比如SOAP）供客户端存取数据。这会在工作中多加一层，但它能够提供额外的安全性，甚至还可能使客户更易整合系统。

话说回来，这有利也有弊，客户端将只能按我们预设或希望其采用的方式获取数据。相比之下，在纯SQL实现中，即使我们没有预料到客户端需要查询最高股价，它们还是可以进行查询。

那么，该采用哪种方案？这里并没有明确的答案。纯文本文件方案或许是一个糟糕的选择，不过，对于SQL或XML方案，不管用不用Web服务，你都可以摆出令人信服的理由。

这类问题的目的不是看你能否得出“正确”答案（并没有唯一正确的答案），而是看你如何设计一个系统，怎么权衡利弊并做出选择。

10.2 你会如何设计诸如Facebook或LinkedIn的超大型社交网站？请设计一种算法，展示两个人之间的“连接关系”或“社交路径”（比如，我 → 鲍勃 → 苏珊 → 杰森 → 你）。（第73页）

解法

这个问题有个不错的解法，就是先移除一些限制条件，解决该问题的简化版本。

步骤 1：简化问题——先忘记有几百万用户

首先，让我们忘掉要应对几百万的用户，针对简单情况设计算法。

我们可以构造一个图，每个人看作一个结点，两个结点之间若有连线，则表示这两个用户为朋友。

```
class Person {
   Person[] friends;
   // 其他信息
}
```

要找到两个人之间的连接，可以从其中一个人开始，直接进行广度优先搜索。

为什么深度优先搜索效果不彰呢？因为它非常低效。两个用户可能只有一度之隔，却可能要在他们的“子树”中搜索几百万个结点后，才能找到这条非常简单而直接的连接。

步骤 2：处理数百万的用户

处理LinkedIn或Facebook这种规模的服务时，不可能将所有数据存放在一台机器上。这就意味着前面定义的简单数据结构`Person`并不管用，朋友的资料和我们的资料不一定在同一台机器上。我们要换种做法，将朋友列表改为他们ID的列表，并按如下方式追踪。

(1) 针对每个朋友ID，找出所在机器的位置：`int machine_index = getMachineIDForUser(personID);`。

(2) 转到编号为`#machine_index`的机器。

(3) 在那台机器上，执行：`Person friend = getPersonWithID(person_id);`。

下面的代码描绘了这一过程。我们定义了一个`Server`类，包含一份所有机器的列表，还有一个`Machine`类，代表一台单独的机器。这两个类都用了散列表，从而有效地查找数据。

```
public class Server {
   HashMap<Integer, Machine> machines =
     new HashMap<Integer, Machine>();
   HashMap<Integer, Integer> personToMachineMap =
     new HashMap<Integer, Integer>();

   public Machine getMachineWithId(int machineID) {
     return machines.get(machineID);
   }

   public int getMachineIDForUser(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     return machineID == null ? -1 : machineID;
   }

   public Person getPersonWithID(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     if (machineID == null) return null;

     Machine machine = getMachineWithId(machineID);
     if (machine == null) return null;

     return machine.getPersonWithID(personID);
```

```java
  }
}

public class Person {
  private ArrayList<Integer> friendIDs;
  private int personID;

  public Person(int id) { this.personID = id; }

  public int getID() { return personID; }
  public void addFriend(int id) { friends.add(id); }
}

public class Machine {
  public HashMap<Integer, Person> persons =
    new HashMap<Integer, Person>();
  public int machineID;

  public Person getPersonWithID(int personID) {
    return persons.get(personID);
  }
}
```

其实还有更多的优化和后续问题有待讨论，下面是其中的一些想法。

优化：减少机器间跳转的次数

从一台机器跳转到另一台机器的开销很昂贵。不要为了找到某个朋友就在机器之间任意跳转，而是试着批处理这些跳转动作。举例来说，如果有五个朋友都在同一台机器上，那就应该一次性找出来。

优化：智能划分用户和机器

人们跟生活在同一国家的人成为朋友的可能性比较大。因此，不要随意将用户划分到不同机器上，而应该尽量按国家、城市、州等进行划分。这样一来，就可以减少跳转的次数。

问题：广度优先搜索通常要求"标记"访问过的结点。在这种情况下你会怎么做？

在广度优先搜索中，通常我们会设定结点类的visited标志，以标记访问过的结点。但针对此题，我们并不想这么做。同一时间可能会执行很多搜索操作，因此直接编辑数据的做法并不妥当。反之，我们可以利用散列表模仿结点的标记动作，以查询结点id，看它是否访问过。

其他扩展问题

- 在真实世界中，服务器会出故障。这会对你造成什么影响？
- 你会如何利用缓存？
- 你会一直搜索，直到图的终点（无限）吗？该如何判断何时放弃？
- 在现实生活中，有些人比其他人拥有更多朋友的朋友，因此更容易在你和其他人之间构建一条路径。该如何利用该数据选择从哪里开始遍历？

这些只是你或者面试官可能会提出的部分扩展问题，其实还有其他许多问题可以深入讨论。

10.3　给定一个输入文件，包含 40 亿个非负整数，请设计一种算法，产生一个不在该文件中的整数。假定你有 1GB内存来完成这项任务。

进阶

如果只有 10MB内存可用，该怎么办？（第 73 页）

解法

总共可能有2^{32}或40亿个不同的整数，其中非负整数共2^{31}个。我们可以使用1GB内存，或者80亿个比特。

这样一来，用这80亿个比特，就可以将所有整数映射到可用内存的不同比特位，处理逻辑如下。

(1) 创建包含40亿个比特的位向量（BV，bit vector）。回想一下，位向量其实就是数组，利用整数（或另一种数据类型）数组紧凑地储存布尔值。每个整数可存储一串32比特或布尔值。

(2) 将BV的所有元素初始化为0。

(3) 扫描文件中的所有数字（num），并调用`BV.set(num, 1)`。

(4) 接着，再次从索引0开始扫描BV。

(5) 返回第一个值为0的索引。

下面的代码示范了上面的算法。

```
long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
byte[] bitfield = new byte [(int) (numberOfInts / 8)];
void findOpenNumber() throws FileNotFoundException {
   Scanner in = new Scanner(new FileReader("file.txt"));
   while (in.hasNextInt()) {
     int n = in.nextInt ();
     /* 使用OR操作符设置一个字节的第n位，
      * 找出bitfield中相对应的数字，
      * （例如，10将对应于字节数组中索引2
      * 的第2位） */
     bitfield [n / 8] |= 1 << (n % 8);
   }

   for (int i = 0; i < bitfield.length; i++) {
     for (int j = 0; j < 8; j++) {
        /* 取回每个字节的各个比特。当发现
         * 某个比特为0时，即找到相对应的值 */
        if ((bitfield[i] & (1 << j)) == 0) {
           System.out.println (i * 8 + j);
           return;
        }
     }
   }
}
```

进阶：只能使用10MB内存该怎么办？

对数据集进行两次扫描，就可以找出不在文件中的整数。我们可以将全部整数划分成同等大小的区块（稍后会讨论如何决定大小）。这里假设要将整数划分为大小为1000的区块。那么，区

块0代表0 ~ 999的数字，区块1代表1000 ~ 1999的数字，依此类推。

因为所有数值各不相同，我们很清楚每个区块应该有多少数字，所以，扫描文件时，数一数0 ~ 999之间有多少个值，1000 ~ 1999之间有多少个值，依此类推。如果在某个区块内只有999个值，即可断定该范围内少了某个数字。

在第二次扫描时，我们要真正找出该范围内少了哪个数字。我们可以采用先前位向量的做法，并忽略该范围之外的任意数字。

眼下，问题在于区块多大才合适？下面先定义若干变量。

- 令`rangeSize`为第一次扫描时每个区块的范围大小。
- 令`arraySize`表示第一次扫描时区块的个数。注意，`arraySize` = 2^{32} / `rangeSize`，因为一共有2^{32}个整数。

我们需要为`rangeSize`选择一个值，以使第一次扫描（数组）与第二次扫描（位向量）所需的内存够用。

- 第一次扫描：数组

第一次扫描所需的数组可以填入10MB或大约2^{23}字节的内存中。数组中每个元素均为整数（int），而每个整数有4字节，因此可以使用最多包含约2^{21}个元素的数组。综上，我们可以导出如下式子：

- 第二次扫描：位向量

$$\texttt{arraySize} = \frac{2^{32}}{\texttt{rangeSize}} \leqslant 2^{21}$$

$$\texttt{rangeSize} \geqslant \frac{2^{32}}{2^{21}}$$

$$\texttt{rangeSize} \geqslant 2^{11}$$

我们需要有足够的空间储存`rangeSize`个比特。我们可以将2^{23}个字节放进内存，自然就能存放2^{26}个比特。因此，可以推出如下式子：

$$2^{11} \leqslant \texttt{rangeSize} \leqslant 2^{26}$$

在这些条件下，我们有足够的空间回旋，但是如果挑选出越靠近中间的值，那么，在任何时候所需的内存就越少。

下面的代码提供了该算法的一种实现。

```
int bitsize = 1048576; // 2^20比特（2^17字节）
int blockNum = 4096; // 2^12
byte[] bitfield = new byte[bitsize/8];
int[] blocks = new int[blockNum];

void findOpenNumber() throws FileNotFoundException {
   int starting = -1;
   Scanner in = new Scanner (new FileReader (“file.txt”));
   while (in.hasNextInt()) {
       int n = in.nextInt();
       blocks[n / (bitfield.length * 8)]++;
   }
```

```

    for (int i = 0; i < blocks.length; i++) {
      if (blocks[i] < bitfield.length * 8){
         /* 若value < 2^20，那么该区段里至少
          * 少了一个数字 */
         starting = i * bitfield.length * 8;
         break;
      }
    }

    in = new Scanner(new FileReader("file.txt"));
    while (in.hasNextInt()) {
      int n = in.nextInt();
      /* 若该数字落在少数字的区块里，
       * 则记下该数字 */
      if (n >= starting && n < starting + bitfield.length * 8) {
         bitfield [(n-starting) / 8] |= 1 << ((n - starting) % 8);
      }
    }

    for (int i = 0 ; i < bitfield.length; i++) {
      for (int j = 0; j < 8; j++) {
         /* 取回每个字节的各个比特，当发现有比特为0时，
          * 找到相对应的值 */
         if ((bitfield[i] & (1 << j)) == 0) {
            System.out.println(i * 8 + j + starting);
            return;
         }
      }
    }
}
```

紧接着，面试官可能还会问你，可用内存更少的话，又该怎么办？在这种情况下，我们会采用第一步骤的做法重复扫描。首先检查每100万个元素序列中会找到多少个整数。接着，在第二次扫描时，检查每1000个元素的序列中可找到多少个整数。最后，在第三次扫描时，使用位向量找出不在文件中的那个数字。

10.4　给定一个数组，包含 1 到*N*的整数，*N*最大为 32 000，数组可能含有重复的值，且*N*的取值不定。若只有 4KB内存可用，该如何打印数组中所有重复的元素。（第 73 页）

解法

我们有4KB内存可用，也就是最多可寻址$8 * 4 * 2^{10}$个比特。注意，$32 * 2^{10}$要比32 000大。我们可以创建含有32 000个比特的位向量，其中每个比特代表一个整数。

利用这个位向量，就可以迭代访问整个数组，发现数组元素*v*时，就将位*v*设定为1。碰到重复元素时，就打印出来。

```
public static void checkDuplicates(int[] array) {
   BitSet bs = new BitSet(32000);
   for (int i = 0; i < array.length; i++) {
     int num = array[i];
     int num0 = num - 1; // bitset从0开始，数字从1开始
```

```
    if (bs.get(num0)) {
      System.out.println(num);
    } else {
      bs.set(num0);
    }
  }
}

class BitSet {
  int[] bitset;

  public BitSet(int size) {
    bitset = new int[size >> 5]; // 除以32
  }

  boolean get(int pos) {
    int wordNumber = (pos >> 5); // 除以32
    int bitNumber = (pos & 0x1F); // 除以32取余数
    return (bitset[wordNumber] & (1 << bitNumber)) != 0;
  }

  void set(int pos) {
    int wordNumber = (pos >> 5); // 除以32
    int bitNumber = (pos & 0x1F); // 除以32取余数
    bitset[wordNumber] |= 1 << bitNumber;
  }
}
```

注意，虽然此题不太难，但重要的是实现代码要写得干净利落。这也是为什么要定义位向量类来保存大型的位向量。要是面试官允许（也可能不会），那就可以使用Java内置的`BitSet`类。

10.5 如果要设计一个网络爬虫程序，该怎么避免陷入无限循环？（第73页）

解法

针对此题，第一个要问自己的是：什么情况下才会出现无限循环？最直接的答案是，如果将整个网络想象成一个链接的图，图中有环就会出现无限循环。

为了避免无限循环，我们只需检测有没有环。一种做法是创建一个散列表，访问过页面v后，将`hash[v]`设为真（`true`）。

这种解法意味着使用广度优先搜索的方式抓取网站。每访问一个页面，我们就会收集它的所有链接，并将它们插入队列末尾。若发现某个页面已访问，就将其忽略。

这个方法不错，不过访问页面v意味着什么？页面v是基于它的内容还是URL来定义的？

如果页面是根据其URL定义的，我们必须认识到URL参数可能代表完全不同的页面。例如，页面`www.careercup.com/page?id=microsoft-interview-questions`与页面`www.careercup.com/page?id=google-interview-questions`是完全不一样的。不过，只要URL参数不是Web应用识别和处理的，就可以将它附加到任意URL之后，而不会真的改变页面，比如，页面`www.careercup.com?foobar=hello`与`www.careercup.com`是一样的。

“好吧，”你或许会说，“那我们就以内容定义页面。”乍一听，似乎还不错，但这并不切实可行。假设careercup.com首页的部分内容是随机生成的。每次访问首页时，它都是不同的页面吗？不见得。

现实情况是目前还没有完美的方式来定义“不同的”页面，这就是此题棘手的地方。

一种解决方法是评估相似程度。根据内容和URL，若某个页面与其他页面具有一定的相似度，则降低抓取其子页面的优先级。对于每个页面，我们都会根据内容片段和页面的URL，算出某种特征码。

下面我们来看看这是如何实现的。

我们有一个数据库，储存了待抓取的一系列条目。每一次循环，我们都会选择最高优先级的页面进行抓取，接着执行以下步骤。

(1) 打开该页面，根据页面的特定片段及其URL，创建该页面的特征码。

(2) 查询数据库，看看最近是否已抓取拥有该特征码的页面。

(3) 若有此特征码的页面最近已被抓取过，则将该页面插回数据库，并调低优先级。

(4) 若未抓取，则抓取该页面，并将它的链接插入数据库。

根据上面的实现，我们怎么也“完不成”整个Web的抓取，但可以避免陷入页面循环的情况。若想最终“完成”整个Web的抓取（显然，只有当这个“Web”是诸如企业内部网那种较小的系统时才可行），那么，可以设定一个保证页面一定会被抓取的最低优先级。

这只是一个简化的解法，实际上还有许多其他同样有效的解法。这类问题更像是你跟面试官之间的对话，可能引发出各种各样的讨论。事实上，针对此题的讨论很有可能引出下一题。

10.6　给定 100 亿个网址，如何检测出重复的文件？这里所谓的”重复“是指两个URL完全相同。（第 73 页）

解法

100亿个网址（URL）要占用多少空间呢？如果每个网址平均长度为100个字符，每个字符占4字节，则这份100亿个网址的列表将占用约4兆兆字节（4TB）。在内存中可能放不下那么多数据。

不过，不妨假装一下，这些数据真的奇迹般地放进了内存，毕竟先求解简化的题目是很有用的做法。对于此题的简化版，只要创建一个散列表，若在网址列表中找到某个URL，就映射为`true`。（另一种做法是对列表进行排序，找出重复项，这需要额外耗费一些时间，但几无优点可言。）

至此，我们得到此题简化版的解法，那么，假设我们手上有4000GB的数据，而且无法全部放进内存，该怎么办？倒也好办，我们可以将部分数据储存至磁盘，或者将数据分拆到多台机器上。

解法 1：储存至磁盘

若将所有数据储存在一台机器上，可以对数据进行两次扫描。第一次扫描是将网址列表拆分

为4000组，每组1GB。简单的做法是将每个网址u存放在名为<x>.txt的文件中，其中x = hash(u) % 4000。也就是说，我们会根据网址的散列值（除以分组数量取余数）分割这些网址。这样一来，所有散列值相同的网址都会位于同一文件。

第二次扫描时，我们其实是在实现前面简化版问题的解法：将每个文件载入内存，创建网址的散列表，找出重复的。

解法 2：多台机器

另一种解法的基本流程是一样的，只不过要使用多台机器。在这种解法中，我们会将网址发送到机器x上，而不是储存至文件<x>.txt。

使用多台机器有优点也有缺点。

主要优点是可以并行执行这些操作，同时处理4000个分组。对于海量数据，这么做就能迅速有效地解决问题。

缺点是现在必须依靠4000台不同的机器，同时要做到操作无误。这可能不太现实（特别是对于数据量更大、机器更多的情况），我们需要开始考虑如何处理机器故障。此外，涉及这么多机器，无疑大幅增加了系统的复杂性。

话说回来，这两种解法都不错，都值得与面试官讨论一番。

10.7 想象有个Web服务器，实现简化版搜索引擎。这套系统有 100 台机器来响应搜索查询，可能会对另外的机器集群调用processSearch(string query)以得到真正的结果。响应查询请求的机器是随机挑选的，因此两个同样的请求不一定由同一台机器响应。方法processSearch的开销很大，请设计一种缓存机制，缓存最近几次查询的结果。当数据发生变化时，务必说明该如何更新缓存。**（第 73 页）**

解法

在开始设计系统之前，必须先理解此题的真正含义。如我们所预料的，这类题目有很多细节都比较模糊。为了提供一个解法，我们将做出一些合理的假设，不过，你应该与面试官深入讨论这些细节。

假设

下面是针对这个解法做出的几个假设条件。基于系统设计和解题的方法，你可能还会做出其他假设条件。记住，虽然某些方法会比其他的好一些，但并没有唯一“正确”的方法。

- 除了必要时往外调用processSearch，所有查询处理都在最初被调用的那台机器上完成。
- 我们希望缓存的搜索查询数量庞大（几百万）。
- 机器之间的调用速度相对较快。
- 给定查询的结果是一个有序的网址列表，每个网址关联50个字符的标题和200个字符的摘要。
- 最常见的查询非常热门，以至于它们总是会存在缓存中。

重申一次，这些不是唯一的有效假设，仅是其中几个合理的假设。

系统需求

设计缓存机制时，显然我们需要支持两个主要功能：

- 给定某个键，快速有效地查找出来；
- 旧的数据会过期，从而让它可被新的数据取代。

此外，当某次查询的结果改变时，我们还必须处理缓存的更新或清除。因为有些查询非常常见，有可能长驻在缓存中，我们不能干等着该数据过期。

步骤 1：设计单系统的缓存

此题有个好解法：先针对单台机器设计缓存。那么，又该创建什么样的数据结构，使我们得以轻易清除旧数据，还能高效地根据键查找出相对应的值？

- 使用链表可以轻易清除旧数据，只需将“新鲜”项移到链表前方。当链表超过一定大小时，我们可以删除链表末尾的元素。
- 散列表可以高效查找数据，但通常无法轻易地清除数据。

怎样才能做到两全其美呢？将这两种数据结构融合在一起即可，下面是具体做法。

跟之前一样创建一个链表，每次访问结点后，这个结点就会移至链表首部。这样一来，链表尾部将总是包含最陈旧的信息。

此外，还需要一个散列表，将查询映射为链表中相应的结点。这样不仅可以有效返回缓存的结果，还能将适当的结点移至链表首部，从而更新其“新鲜度”。

为了说明这种方法，下面给出了缩略的缓存实现代码。本书网站提供了这些代码的完整版本。注意，在面试中，一般不会要求你为此写出完整的代码，也不会要求你设计更大的系统。

```java
public class Cache {
   public static int MAX_SIZE = 10;
   public Node head, tail;
   public HashMap<String, Node> map;
   public int size = 0;

   public Cache() {
     map = new HashMap<String, Node>();
   }

   /* 将结点移至链表前方 */
   public void moveToFront(Node node) { ... }
   public void moveToFront(String query) { ... }

   /* 从链表中移除结点 */
   public void removeFromLinkedList(Node node) { ... }

   /* 从缓存中获取结果，并更新链表 */
   public String[] getResults(String query) {
     if (!map.containsKey(query)) return null;

     Node node = map.get(query);
     moveToFront(node); // 更新新鲜度
     return node.results;
```

```
    }

    /* 将结果插入链表，并散列 */
    public void insertResults(String query, String[] results) {
      if (map.containsKey(query)) { // 更新值
         Node node = map.get(query);
         node.results = results;
         moveToFront(node); // 更新新鲜度
         return;
      }

      Node node = new Node(query, results);
      moveToFront(node);
      map.put(query, node);

      if (size > MAX_SIZE) {
         map.remove(tail.query);
         removeFromLinkedList(tail);
      }
    }
}
```

步骤 2：扩展到多台机器

现在，我们了解了如何设计单台机器的缓存，接下来还需了解，当查询被发送至许多不同的机器时，如何设计缓存。回想一下问题描述：不能保证某个查询一定会发送给同一台机器。

首先，我们需要决定缓存跨机器共享到什么程度。有以下几种选择可供参考。

- 选择1：每台机器都有自己的缓存

简单的选择是每台机器都有自己的缓存。也就是说，如果“foo”在短时间内被发送给机器1两次，在第二次，结果会从缓存中返回。但是，如果“foo”先发送给机器1然后发送至机器2，则两次都会被视作全新的查询。

这么做的优点是相对快速，因为不涉及机器之间的调用。可惜，由于许多重复查询都会被视作全新查询，作为优化工具的缓存并不是那么有效。

- 选择2：每台机器都有一个缓存的副本

另一个极端是，我们可以给每台机器一个缓存的完整副本。当新的条目添加至缓存时，它们会被发送给所有机器。包括链接和散列表在内的整个数据结构都会被复制。

这种设计意味着常见的查询几乎总是会在缓存里，因为所有机器的缓存都是相同的。但是，主要的缺点是更新缓存意味着要将数据发送给N台机器，其中N是响应集群的规模。此外，每个条目占用的空间是上一种做法的N倍，因此缓存所能存放的数据要少得多。

- 选择3：每台机器储存一部分缓存

第三种选择是将缓存分割开，每台机器存放缓存的不同部分。然后，当机器i需要查找某次查询的结果时，它会算出哪一台机器持有这个值，接着请求这台机器（机器j）在它的缓存里查找该查询。

但是，机器i怎么知道哪一台机器持有这部分散列表？

一种选择是根据算式hash(query) % N指定查询的结果。然后，机器i只需利用这个算式即可得出储存结果的机器j。

因此，当新的查询进入机器i时，这台机器会应用上面的算式从而调用机器j。随后，机器j会从它的缓存中返回待查询的值，或者调用`processSearch(query)`得到结果。机器j会更新其缓存，并将结果返回给机器i。

或者，你也可以这样设计系统：机器j在其当前缓存中找不到查询的结果，则直接返回`null`。这就要求机器i调用`processSearch`，然后将结果转发给机器j储存。这个实现实际上会增加机器与机器间的调用数量，没什么优势可言。

步骤 3：内容改变时更新结果

回想一下，有些查询可能非常热门，以致缓存足够大的话，它们可能会永久存在缓存中。当某些内容改变时，我们需要通过某种机制来定期或“按需”刷新缓存的结果。

要回答这个问题，我们需要考虑结果何时才会改变（最好跟面试官讨论一下）。结果改变的主要时机如下。

(1) 网址对应的内容变了（或网址对应的页面被移除）。

(2) 为反映页面排名变化，搜索结果的排序也变了。

(3) 特定查询出现了新页面。

为了处理情况(1)和情况(2)，可以另外创建一个散列表，指示哪个缓存查询与特定网址关联。这些缓存可以完全独立于其他缓存进行处理，并放在不同的机器上。不过，这种解法可能需要大量的数据。

另外，如果数据不要求即时刷新（一般来说不需要），我们可以定期遍历每台机器上储存的缓存，将与更新过的网址相关联的结果清除掉。

情况(3)很难处理。我们可以通过解析新网址对应的内容并从缓存中清除这些单一词的查询，来更新单一词查询。不过，这仅能处理单一词的查询。

情况(3)（或我们要处理的其他类似情况）有个不错的处理方式，就是实现缓存的“自动逾期”。也就是说，我们会强加一个超时，任何一个查询，不管它有多热门，都无法在缓存中存放超过x分钟。这将确保所有的数据都会定期刷新。

步骤 4：继续改进

根据你做出的假设和想要优化的情况，这个设计还可以有不少可改进和优化之处。

其中有个优化是更好地支持有些查询非常热门的情况。例如，假设（举个极端的例子）所有查询中，有1%都含有某个字符串。那么，机器i不必每次都将这个搜索请求转给机器j，应该只向`j`转发一次，然后机器`i`就可以直接将结果储存在自己的缓存中。

或者，我们还可以重新架构整个系统，根据查询的散列值而不是随机将查询分配给某台机器（由此也得到缓存的位置）。不过，这么做也有利有弊。

另一个优化是针对“自动过期”机制的。按照前面的描述，这个机制会在X分钟后清除任意数据。然而，相比其他数据（如历史股价），我们希望某些数据（如时事新闻）的更新更频繁，

可以根据主题或网址实现不同的自动逾期机制。对于后一种情况，根据页面以往的更新频度，每个网址会设置不同的超时值。该搜索查询的超时值是每个网址超时值的最小值。

这只是一部分可以改进的地方。记住，这类题型并没有唯一正确的解法，其用意是让你与面试官讨论设计准则，展示你的思考方式和解题方法。

9.11　排序与查找

11.1 给定两个排序后的数组A和B，其中A的末端有足够的缓冲空容纳B。编写一个方法，将B合并入A并排序。**（第 77 页）**

解法

已知数组A末端有足够的缓冲，不需要再分配额外空间。程序的处理逻辑很简单，就是逐一比较A和B中的元素，并按顺序插入数组，直至耗尽A和B中的所有元素。

这么做的唯一问题是，如果将元素插入数组A的前端，就必须将原有的元素往后移动，以腾出空间。更好的做法是将元素插入数组A的末端，那里都是空闲的可用空间。

下面的代码就实现了上述做法，从数组A和B的末端元素开始，将最大的元素放到数组A的末端。

```
public static void merge(int[] a, int[] b, int lastA, int lastB) {
   int indexA = lastA - 1; /* 数组a最后元素的索引 */
   int indexB = lastB - 1; /* 数组b最后元素的索引 */
   int indexMerged = lastB + lastA - 1; /* 合并后数组的最后元素索引 */

   /* 合并a和b，从这两个数组的最后元素开始 */
   while (indexA >= 0 && indexB >= 0) {
     /* 数组a最后元素 > 数组b最后元素 */
     if (a[indexA] > b[indexB]) {
        a[indexMerged] = a[indexA]; // 复制元素
        indexMerged--; // 更新索引
        indexA--;
     } else {
        a[indexMerged] = b[indexB]; // 复制元素
        indexMerged--; // 更新索引
        indexB--;
     }
   }

   /* 将数组b剩余元素复制到适当的位置 */
   while (indexB >= 0) {
     a[indexMerged] = b[indexB];
     indexMerged--;
     indexB--;
   }
}
```

注意，处理完B的剩余元素后，你不需要复制A的剩余元素，因为这些元素已经在那里了。

11.2　编写一个方法，对字符串数组进行排序，将所有变位词[1]排在相邻的位置。（第 77 页）

解法

此题有个要求，对数组中的字符串进行分组，将变位词排在一起。注意，除此之外，并没有要求这些词按特定顺序排列。

做法之一就是套用一种标准排序算法，比如归并排序或快速排序，并修改比较器（comparator）。这个比较器用来指示两个字符串互为变位词就是相等的。

检查两个词是否为变位词，最简单的方法是什么呢？我们可以数一数每个字符串中各个字符出现的次数，两者相同则返回`true`。或者，直接对字符串进行排序，若两个字符串互为变位词，排序后就相同。

比较器的实现代码如下。

```
public class AnagramComparator implements Comparator<String> {
  public String sortChars(String s) {
    char[] content = s.toCharArray();
    Arrays.sort(content);
    return new String(content);
  }

  public int compare(String s1, String s2) {
    return sortChars(s1).compareTo(sortChars(s2));
  }
}
```

下面，利用这个`compareTo`方法而不是一般的比较器对数组进行排序。

```
Arrays.sort(array, new AnagramComparator());
```

这个算法的时间复杂度为$O(n \log(n))$。

这可能是使用通用排序算法所能取得的最佳情况了，但实际上，并不需要对整个数组进行排序，只需将变位词分组放在一起即可。

我们可以使用散列表做到这一点，这个散列表会将排序后的单词映射到它的一个变位词列表。举例来说，`acre`会映射到列表`{acre, race, care}`。一旦将所有同为变位词的单词分组在一起，就可以将它们放回到数组中。

下面是该算法的实现代码。

```
public void sort(String[] array) {
  Hashtable<String, LinkedList<String>> hash =
    new Hashtable<String, LinkedList<String>>();

  /* 将同为变位词的单词分在同一组 */
  for (String s : array) {
    String key = sortChars(s);
    if (!hash.containsKey(key)) {
       hash.put(key, new LinkedList<String>());
```

① 由变换某个词或短语的字母顺序构成的新的词或短语。例如，“triangle”是“integral”的变位词。——译者注

```
      }
      LinkedList<String> anagrams = hash.get(key);
      anagrams.push(s);
    }

    /* 将散列表转换为数组 */
    int index = 0;
    for (String key : hash.keySet()) {
      LinkedList<String> list = hash.get(key);
      for (String t : list) {
         array[index] = t;
         index++;
      }
    }
}
```

你或许看出来了，上面的算法是从桶排序法修改而来的。

11.3 给定一个排序后的数组，包含*n*个整数，但这个数组已被旋转过很多次，次数不详。请编写代码找出数组中的某个元素。可以假定数组元素原先是按从小到大的顺序排列的。（第77页）

解法

你是不是觉得此题要用到二分查找法？没错。

在经典二分查找法中，我们会将*x*与中间元素进行比较，以确定*x*属于左半部分还是右半部分。此题的复杂之处在于数组被旋转过了，可能有一个拐点。以下面两个数组为例：

```
Array1: {10, 15, 20,  0,  5}
Array2: {50,  5, 20, 30, 40}
```

这两个数组的中间元素都是20，但5在其中一个数组的左边，在另一个的右边。因此，只将*x*与中间元素进行比较是不够的。

不过，如果再仔细观察一下，就会发现数组有一半（左边或右边）必定是按正常顺序（升序）排列的。因此，我们可以看看按正常顺序排列的那一半数组，确定应该搜索左半边还是右半边。

例如，如果要在`Array1`中查找5，我们可以比较左侧元素（10）和中间元素（20）。由于10 < 20，左半边一定是按正常顺序排列的。另外，由于5不在这两个元素之间，因此接下来应该搜索右半边。

在`Array2`中，可以看到50 > 20，因此右半边必定是按正常顺序排列的。接着查看中间元素（20）和右侧元素（40），检查5是否落在这两个元素之间。显然5并不落在两者之间，因此接下来要搜索右半边。

如果左侧元素和中间元素完全相同，比如数组`{2, 2, 2, 3, 4, 2}`，这种情况就比较复杂了。这里我们可以检查最右边的元素是否不同。若不同，可以只搜索右半边，否则，两边都得搜索。

```
public int search(int a[], int left, int right, int x) {
   int mid = (left + right) / 2;
   if (x == a[mid]) { // 找到元素
```

```
      return mid;
    }
    if (right < left) {
      return -1;
    }

    /* 左半边或右半边必有一边是按正常顺序排列，
     * 找出是哪一半边，然后利用按正常顺序排列的
     * 半边，确定该搜索哪一边 */
    if (a[left] < a[mid]) { // 左半边为正常排序
      if (x >= a[left] && x <= a[mid]) {
         return search(a, left, mid - 1, x); // 搜索左半边
      } else {
         return search(a, mid + 1, right, x); // 搜索右半边
      }
    } else if (a[mid] < a[left]) { // 右半边为正常排序
      if (x >= a[mid] && x <= a[right]) {
         return search(a, mid + 1, right, x); // 搜索右半边
      } else {
         return search(a, left, mid - 1, x); // 搜索左半边
      }
    } else if (a[left] == a[mid]) { // 左半边都是重复元素
      if (a[mid] != a[right]) { // 若右边元素不同，则搜索那一边
         return search(a, mid + 1, right, x); // 搜索右半边
      } else { // 否则，两边都得搜索
         int result = search(a, left, mid - 1, x); // 搜索左半边
         if (result == -1) {
            return search(a, mid + 1, right, x); // 搜索右半边
         } else {
            return result;
         }
      }
    }
    return -1;
}
```

若所有元素都不同，则上述代码执行的时间复杂度为$O(\log n)$。有很多元素重复的话，算法时间复杂度则为$O(n)$。因为若有很多重复元素，数组（或子数组）的左半边和右半边往往都得查找。

注意，尽管此题并不是太难理解，但要完美无瑕地实现却很难。实现时难免会犯错，不必太自责。因为很容易就犯差一错误和其他不易察觉的错误，所以，务必对代码进行全面彻底的测试。

11.4　设想你有一个 20GB的文件，每一行一个字符串。请说明将如何对这个文件进行排序。（第 77 页）

解法

当面试官给出20GB大小的限制时，实际上在暗示些什么。就此题而言，这表明他们不希望你将数据全部载入内存。

该怎么办呢？做法是只将部分数据载入内存。

我们将把整个文件划分成许多块，每个块x MB，其中x是可用的内存大小。每个块各自进行排序，然后存回文件系统。

各个块一旦完成排序，我们便将这些块逐一合并在一起，最终就能得到全都排好序的文件。

这个算法被称为外部排序（external sort）。

11.5 有个排序后的字符串数组，其中散布着一些空字符串，编写一个方法，找出给定字符串的位置。**（第 77 页）**

解法

如果没有那些空字符串，就可以直接使用二分查找法。比较待查找字符串`str`和数组的中间元素，然后继续搜索下去。

针对数组中散布一些空字符串的情形，我们可以对二分查找法稍作修改，所需的修改就是与`mid`进行比较的地方，如果`mid`为空字符串，就将`mid`换到离它最近的非空字符串的位置。

下面以递归方式解决此题，稍加修改，就可以迭代方式实现。本书可下载的代码里提供了迭代实现。

```
public int searchR(String[] strings, String str, int first,
                   int last) {
   if (first > last) return -1;
   /* 将mid移到中间 */
   int mid = (last + first) / 2;

   /* 若mid为空字符串，找出离它最近的非空字符串 */
   if (strings[mid].isEmpty()) {
     int left = mid - 1;
     int right = mid + 1;
     while (true) {
        if (left < first && right > last) {
           return -1;
        } else if (right <= last && !strings[right].isEmpty()) {
           mid = right;
           break;
        } else if (left >= first && !strings[left].isEmpty()) {
           mid = left;
           break;
        }
        right++;
        left--;
     }
   }

   /* 检查字符串，如有必要则继续递归 */
   if (str.equals(strings[mid])) { // 找到了
     return mid;
   } else if (strings[mid].compareTo(str) < 0) { // 搜索右半边
     return searchR(strings, str, mid + 1, last);
   } else { // 搜索左半边
     return searchR(strings, str, first, mid - 1);
```

```
    }
}

public int search(String[] strings, String str) {
    if (strings == null || str == null || str == “”) {
      return -1;
    }
    return searchR(strings, str, 0, strings.length - 1);
}
```

如果要查找空字符串，务必小心对待。我们该找出空字符串的位置（该操作时间复杂度为$O(n)$）？还是应该把这种情形作为错误处理？

很遗憾，这里并没有正确的答案。关于这一点你应该与面试官进行讨论，只需简单地询问一下，就能表明你是个细心的程序员。

11.6　给定$M \times N$矩阵，每一行、每一列都按升序排列，请编写代码找出某元素。（第 77 页）

解法

解法 1

在第一种方法里，我们可以对每一行进行二分查找，以找到元素在哪。该矩阵有M行，搜索每一行用时$O(\log(N))$，因此这个算法的时间复杂度为$O(M \log(N))$。在你开始构思更好的算法之前，这个算法值得向面试官一提。

要设计一个算法，我们先从一个简单的例子开始。

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

假设要查找元素55，该如何找出它在哪儿呢？

只要看看一行或一列的起始元素，我们就能开始推断待查元素的位置。若一列的起始元素大于55，就表示55不可能在那一列，因为起始元素是那一列的最小元素。此外，我们也可推断55不可能在那一列的右边，因为每一列的第一个元素从左到右依次增大。因此，若那一列的起始元素大于待查找的元素x，就能确定我们必须往那一列的左边查找。

对于矩阵的行来说，可以套用同样的逻辑。若某一行的起始元素大于x，就应该往上查找。

同样地，我们也可以从列或行的末端得出类似的结论，若某一列或行的末尾元素小于x，就必须往下（行）或往右（列）查找，这是因为末尾元素必定是最大的元素。

下面我们可以将这些观察到的要点合并成一个解法，观察到的要点包括：

- 若列的开头大于x，那么x位于该列的左边；
- 若列的末端小于x，那么x位于该列的右边；
- 若行的开头大于x，那么x位于该行的上方；

- 若行的末端小于x，那么x位于该行的下方。

我们可以从任意位置开始搜索，不过，让我们从列的起始元素开始。

我们需要从最大的那一列开始，然后向左移动，这意味着第一个要比较的元素是`array[0][c-1]`，其中c为列的数目。将各个列的开头与x（这里为55）进行比较，就会发现x必定位于列0、列1或列2，比较至`array[0][2]`停下来。

这个元素不一定会在完整矩阵的某一列的末端，但会是某个子矩阵的某一列的末端。同样的条件一样适用，`array[0][2]`的值是40，比55小，由此可知必须往下移动。

现在，我们以下面这个子矩阵为例进行说明（灰色方格已被排除了）。

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

我们可以重复套用以上条件和流程找出55。注意，在此只能使用条件1和条件4。

下面是这个排除算法的实现代码。

```java
public static boolean findElement(int[][] matrix, int elem) {
   int row = 0;
   int col = matrix[0].length - 1;
   while (row < matrix.length && col >= 0) {
     if (matrix[row][col] == elem) {
        return true;
     } else if (matrix[row][col] > elem) {
        col--;
     } else {
        row++;
     }
   }
   return false;
}
```

还有别的做法，我们可以运用另一种看起来更像是二分查找法的解法。其中代码要复杂得多，但也用到了很多相同的技巧。

解法 2：二分查找法

让我们再来看个简单的例子。

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

我们希望能够充分利用矩阵行列已排序的条件，以更有效率地找到元素。因此，试着问问自

己，对于某个元素可能位于什么位置，这个矩阵独特的排序属性意味着什么？

我们知道每一行每一列都是已排序的，也就是说元素`a[i][j]`会大于位于行i、列0和列$j-1$之间的元素，并且大于位于列j、行0和行$i-1$之间的元素。

换句话说：

```
a[i][0] <= a[i][1] <= ... <= a[i][j-1] <= a[i][j]
a[0][j] <= a[1][j] <= ... <= a[i-1][j] <= a[i][j]
```

下面以图表说明，其中深灰色元素大于所有浅灰色元素。

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

浅灰色元素也有顺序：每一个都大于它左边的元素，并且大于它上方的元素，因此，根据传递性，深灰色元素比色块里的其他元素都要大。

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

这意味着，若在矩阵里任意画个长方形，其右下角的元素一定是最大的。

同样地，左上角的元素一定是最小的。下图的颜色标示元素的大小顺序（浅灰色<深灰色<黑色）：

15	20	70	85
20	35	80	95
30	55	95	105
40	80	120	120

让我们回到原先的问题：假设要查找值85，若顺着对角线搜索，可找到元素35和95。利用这些信息可知85的位置吗？

15	20	70	85
25	35	80	95
30	55	95	105
40	80	120	120

85不可能位于黑色区域，因为95位于该区域的左上角，也是该方形里最小的元素。85也不可能位于浅灰色区域，因为35位于该方形的右下角，是该方形中最大的元素。

85必定位于两个白色区域之一。

因此，我们将矩阵分为四个区域，以递归方式搜索左下区域和右上区域。这两个区域也会被分成子区域并继续搜索。

注意到对角线是已排序的，因此可以利用二分查找法进行高效的搜索。

下面是该算法的实现代码。

```
public Coordinate findElement(int[][] matrix, Coordinate origin,
                              Coordinate dest, int x) {
   if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
     return null;
   }
   if (matrix[origin.row][origin.column] == x) {
     return origin;
   } else if (!origin.isBefore(dest)) {
     return null;
   }

   /* 将start和end分别设为对角线的起点和终点。
    * 矩阵不一定是正方形，因此对角线的终点也
    * 可能不等于dest */
   Coordinate start = (Coordinate) origin.clone();
   int diagDist = Math.min(dest.row - origin.row,
                           dest.column - origin.column);
   Coordinate end = new Coordinate(start.row + diagDist,
                                   start.column + diagDist);
   Coordinate p = new Coordinate(0, 0);

   /* 在对角线上进行二分查找，找出第一个
    * 比x大的元素 */
   while (start.isBefore(end)) {
     p.setToAverage(start, end);
     if (x > matrix[p.row][p.column]) {
        start.row = p.row + 1;
        start.column = p.column + 1;
     } else {
        end.row = p.row - 1;
        end.column = p.column - 1;
     }
   }

   /* 将矩阵分为四个区域，搜索左下区域和
    * 右上区域 */
   return partitionAndSearch(matrix, origin, dest, start, x);
}

public Coordinate partitionAndSearch(int[][] matrix,
     Coordinate origin, Coordinate dest, Coordinate pivot,
     int elem) {
   Coordinate lowerLeftOrigin =
     new Coordinate(pivot.row, origin.column);
   Coordinate lowerLeftDest =
     new Coordinate(dest.row, pivot.column - 1);
```

```
  Coordinate upperRightOrigin =
    new Coordinate(origin.row, pivot.column);
  Coordinate upperRightDest =
    new Coordinate(pivot.row - 1, dest.column);

  Coordinate lowerLeft =
    findElement(matrix, lowerLeftOrigin, lowerLeftDest, elem);
  if (lowerLeft == null) {
    return findElement(matrix, upperRightOrigin,
                       upperRightDest, elem);
  }
  return lowerLeft;
}

public static Coordinate findElement(int[][] matrix, int x) {
  Coordinate origin = new Coordinate(0, 0);
  Coordinate dest = new Coordinate(matrix.length - 1,
                                   matrix[0].length - 1);
  return findElement(matrix, origin, dest, x);
}

public class Coordinate implements Cloneable {
  public int row;
  public int column;
  public Coordinate(int r, int c) {
    row = r;
    column = c;
  }

  public boolean inbounds(int[][] matrix) {
    return  row >= 0 && column >= 0 &&
            row < matrix.length && column < matrix[0].length;
  }

  public boolean isBefore(Coordinate p) {
    return row <= p.row && column <= p.column;
  }

  public Object clone() {
    return new Coordinate(row, column);
  }

  public void setToAverage(Coordinate min, Coordinate max) {
    row = (min.row + max.row) / 2;
    column = (min.column + max.column) / 2;
  }
}
```

如果你读过上面所有代码，心里会想："我可没办法在面试时写出所有这些代码！"没错，的确无法全部写出。但是，你在任何面试题上的表现都会比照其他求职者进行评估，因此，如果你无法完整写出代码，他们也同样不能。碰到这类棘手的问题时，你未必处于不利的位置。

将一些代码独立出来写成方法，可以增加你的亮点。例如，将partitionAndSearch独立出

来写成一个方法，想勾勒代码的轮廓就要简单许多。之后有时间的话，你可以再回头填充partitionAndSearch的内容。

11.7 有个马戏团正在设计叠罗汉的表演节目，一个人要站在另一人的肩膀上。出于实际和美观的考虑，在上面的人要比下面的人矮一点、轻一点。已知马戏团每个人的高度和重量，请编写代码计算叠罗汉最多能叠几个人。（第 77 页）

解法

去掉此题的“细枝末节”，可以看出真正要考的题目如下。

给定一个列表，每个元素由一对项目组成。找出最长的子序列，其中第一项和第二项均以非递减的顺序排列。

如果套用简单构造法（或模式匹配法），我们就可以将此题视为如何找出数组中的最长递增序列。

1. 子问题：最长递增子序列

如果元素不必保持一样（相对）的顺序，则只需对数组进行排序即可。这么一来，此题就显得太过简单了，因此，让我们假设元素必须保持一样的相对顺序。

通过一个一个地观察数组元素，可以试着推导出递归算法。首先，你需要了解，就算知道了A[0]到A[i]的最长递增子序列，我们也无法得知A[i + 1]和A[i + 2]的答案。这一点由下面这个简单的例子可知：

```
数组：13, 14, 10, 11, 12
Longest(0 through 0)：13
Longest(0 through 1)：13, 14
Longest(0 through 2)：13, 14
Longest(0 through 3)：13, 14  或  10, 11
Longest(0 through 4)：10, 11, 12
```

如果只是试着以最新的解决方案求出Longest(0 through 4)和Longest(0 through 3)，就会找不到最优解。

然而，我们可以换一种不同的递归解法，之前是试着找出从0到*i*的元素的最长递增子序列，现在改为找出以元素*i*结尾的最长递增子序列。继续使用上面的例子，做法如下：

```
数组：13, 14, 10, 11, 12
Longest(ending with A[0])：13
Longest(ending with A[1])：13, 14
Longest(ending with A[2])：10
Longest(ending with A[3])：10, 11
Longest(ending with A[4])：10, 11, 12
```

注意，以A[i]结尾的最长子序列可以通过检查先前全部解法得出，只要将A[i]附加到最长且“有效”的那个序列即可，所谓“有效”是指符合A[i] > list.tail的任意序列。

2. 真正的问题：最长递增子序列，每个元素均为一对项目

现在，我们知道如何找出一串整数的最长递增子序列，就可以很容易地解决真正的问题，只

9

要将一列表演人员按身高排序，然后对体重套用longestIncreasingSubsequence算法即可。

下面是该算法的实现代码。

```
ArrayList<HtWt> getIncreasingSequence(ArrayList<HtWt> items) {
  Collections.sort(items);
  return longestIncreasingSubsequence(items);
}

void longestIncreasingSubsequence(ArrayList<HtWt> array,
      ArrayList<HtWt>[] solutions, int current_index) {
  if (current_index >= array.size() || current_index < 0) return;
  HtWt current_element = array.get(current_index);

  /* 找出可以附加current_element的最长子序列 */
  ArrayList<HtWt> best_sequence = null;
  for (int i = 0; i < current_index; i++) {
    if (array.get(i).isBefore(current_element)) {
      best_sequence = seqWithMaxLength(best_sequence,
                                solutions[i]);
    }
  }

  /* 附加current_element */
  ArrayList<HtWt> new_solution = new ArrayList<HtWt>();
  if (best_sequence != null) {
    new_solution.addAll(best_sequence);
  }
  new_solution.add(current_element);

  /* 加入到列表中，然后递归 */
  solutions[current_index] = new_solution;
  longestIncreasingSubsequence(array, solutions, current_index+1);
}

ArrayList<HtWt> longestIncreasingSubsequence(
    ArrayList<HtWt> array) {
  ArrayList<HtWt>[] solutions = new ArrayList[array.size()];
  longestIncreasingSubsequence(array, solutions, 0);

  ArrayList<HtWt> best_sequence = null;
  for (int i = 0; i < array.size(); i++) {
    best_sequence = seqWithMaxLength(best_sequence, solutions[i]);
  }

  return best_sequence;
}

/* 返回较长的序列 */
ArrayList<HtWt> seqWithMaxLength(ArrayList<HtWt> seq1,
    ArrayList<HtWt> seq2) {
  if (seq1 == null) return seq2;
  if (seq2 == null) return seq1;
  return seq1.size() > seq2.size() ? seq1 : seq2;
```

```
}

public class HtWt implements Comparable {
   /* 声明等 */

   /* 供sort方法使用 */
   public int compareTo( Object s ) {
     HtWt second = (HtWt) s;
     if (this.Ht != second.Ht) {
        return ((Integer)this.Ht).compareTo(second.Ht);
     } else {
        return ((Integer)this.Wt).compareTo(second.Wt);
     }
   }

   /* 若this应该排在other之前，则返回true。
    * 注意，this.isBefore(other)和other.isBefore(this)
    * 两者皆为false。这跟compareTo方法不同，
    * 若a < b，则b > a */
   public boolean isBefore(HtWt other) {
     if (this.Ht < other.Ht && this.Wt < other.Wt) return true;
     else return false;
   }
}
```

这个算法的时间复杂度为$O(n^2)$，确实有个算法的用时可以达到$O(n \log(n))$，但要复杂得多，不太可能在面试中推导出来，哪怕是有提示也办不到。不过，如果你有兴趣试试这种解法，不妨上网搜一下，应该能搜到该解法的不少说明。

11.8 假设你正在读取一串整数。每隔一段时间，你希望能找出数字x的秩（小于或等于x的值的数目）。请实现数据结构和算法支持这些操作。也就是说，实现`track(int x)`方法，每读入一个数字都会调用该方法；以及`getRankOfNumber(int x)`方法，返回值为小于或等于x的元素个数（不包括x本身）。（第 77 页）

解法

有种相对简单的实现方式是用一个数组存放所有已排好序的元素。当有新元素进来时，我们需要搬移其他元素以腾出空间。这么一来，`getRankOfNumber`实现起来会非常有效，只需执行二分查找，返回索引。

然而，插入元素（也就是`track(int x)`函数）将会非常低效，我们需要一种数据结构，不仅能在插入新元素时加以更新，还能维持相对排列顺序。二叉查找树正好适用。

之前是要把元素插入数组，现在则要将元素插入二叉查找树。`track(int x)`方法的时间复杂度为$O(\log n)$，其中n为树的大小（当然，前提为这棵树是平衡的）。

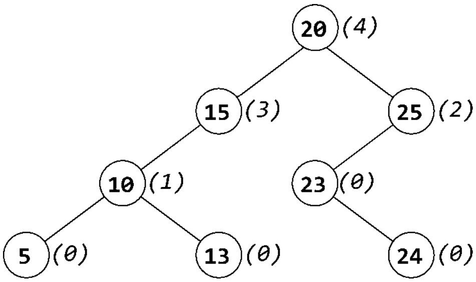
要找出某个数的秩，可以执行中序遍历，并在访问结点时利用计数器记录数量。目标是找到x时，计数器变量将会是小于x的元素的数量。

在查找x期间，只要向左移动，计数器变量就不会变，为什么呢？因为右边跳过的所有值都比x大。毕竟最小的元素（秩为1）是最左边的结点。

可是当向右移动时，我们跳过了左边的一堆元素。这些元素都比x小，因此，必须增加计数器的值，这个值等于左子树的元素个数。

我们不会去计算左子树的大小（效率低），而是在加入新元素时，记录相关信息。

接下来将以下面的树为例说明。在下图中，括号内的数字代表左子树的结点数量（或者，换句话说，该结点相对于它的子树的秩）。



假设我们想知道24在上面这棵树中的秩，会先将24与根结点20比较，发现24位于右边。根结点的左子树有4个结点，再加上根结点本身，总共有5个结点小于24，因此我们会将计数器变量counter设为5。

然后，将24与结点25进行比较，发现24必定位于左边。counter变量的值不会更新，因为我们并未“跳过”任何较小的结点，counter变量的值仍为5。

接着，将24与结点23进行比较，发现24必定位于右边。counter变量会增加1（变为6），因为23没有左边的结点。

最后，我们找到24并返回counter值：6。

这个递归算法如下：

```
int getRank(Node node, int x) {
   if x is node.data
     return node.leftSize()
   if x is on left of node
     return getRank(node.left, x)
   if x is on right of node
     return node.leftSize() + 1 + getRank(node.right, x)
}
```

下面是完整的代码。

```
public class Question {
   private static RankNode root = null;

   public static void track(int number) {
     if (root == null) {
        root = new RankNode(number);
     } else {
        root.insert(number);
     }
   }
```

```java

   public static int getRankOfNumber(int number) {
     return root.getRank(number);
   }

   ...
}

public class RankNode {
   public int left_size = 0;
   public RankNode left, right;
   public int data = 0;
   public RankNode(int d) {
     data = d;
   }

   public void insert(int d) {
     if (d <= data) {
        if (left != null) left.insert(d);
        else left = new RankNode(d);
        left_size++;
     } else {
        if (right != null) right.insert(d);
        else right = new RankNode(d);
     }
   }

   public int getRank(int d) {
     if (d == data) {
        return left_size;
     } else if (d < data) {
        if (left == null) return -1;
        else return left.getRank(d);
     } else {
        int right_rank = right == null ? -1 : right.getRank(d);
        if (right_rank == -1) return -1;
        else return left_size + 1 + right_rank;
     }
   }
}
```

注意上面的代码是怎么处理*d*不在树里的情况的。我们会检查返回值是否为–1，当发现为–1时，将它往上返回。你必须处理诸如此类情况，这很重要。

9.12 测试

12.1 找出以下代码中的错误（可能不止一处）:

```c
unsigned int i;
for (i = 100; i >= 0; --i)
   printf(“%d\n”, i);
```

（第82页）

解法

这段代码有两处错误。

首先，根据定义，`unsigned int`类型的变量一定会大于或等于零。因此，`for`循环的测试条件一直为真，将陷入无限循环。

要打印100到1之间的所有整数，正确的做法是测试$i > 0$。如果真的想打印0，可以在`for`循环之后加一条`printf`语句。

```
unsigned int i;
for (i = 100; i > 0; --i)
   printf("%d\n", i);
```

另一个需要修正的地方是用`%u`代替`%d`，因为这里打印的是`unsigned int`型变量。

```
unsigned int i;
for (i = 100; i > 0; --i)
   printf("%u\n", i);
```

现在，这段代码会正确地打印100到1的整数序列（按降序排列）。

12.2　有个应用程序一运行就崩溃，现在你拿到了源码。在调试器中运行 10 次之后，你发现该应用每次崩溃的位置都不一样。这个应用只有一个线程，并且只调用C标准库函数。究竟是什么样的编程错误导致程序崩溃？该如何逐一测试每种错误？**（第 83 页）**

解法

具体如何处理这个问题要视待诊断应用程序的类型而定。不过，我们还是可以给出一些随机崩溃的常见原因。

(1) **随机变量**：该应用程序可能用到某个随机变量或可变分量，程序每次执行时取值不定。具体的例子包括用户输入、程序生成的随机数，或当前时间等。

(2) **未初始化变量**：该应用程序可能包含一个未初始化变量，在某些语言中，该变量可能含有任意值。这个变量取不同值可能导致代码每次执行路径有所不同。

(3) **内存泄漏**：该程序可能存在内存溢出。每次运行时引发问题的可疑进程随机不定，这与当时运行的进程数量有关。另外还包括堆溢出或栈内数据被破坏。

(4) **外部依赖**：该程序可能依赖别的应用程序、机器或资源。要是存在多处依赖，程序就有可能在任意位置崩溃。

为了找出问题的原因，我们首先应该尽可能地了解这个应用程序。谁在运行这个程序？他们用它做什么？这个程序属于哪种应用？

此外，尽管应用程序每次崩溃的位置不尽相同，但还是有办法确定它可能与特定组件或场景有关。例如，有可能只是启动该应用程序而不进行其他操作时，这个程序从不崩溃。它只有在载入文件之后的某个时间点才会崩溃。或者，有可能每次崩溃都出现在底层组件如文件I/O上。

要解决这个问题，消除法也许值得一试。首先，关闭系统中其他所有应用，仔细追踪资源使用。如果该程序有些部分可以关掉，那就设法关掉。在另一台机器上运行该程序，看看能否重现

同一问题。我们可以消除（或修改）的越多，就越容易定位原因。

此外，我们还可以借助工具检查特定情况。例如，要排查前面第二个原因，我们可以利用运行时工具来检查未初始化变量。

这些问题不仅考查你解决问题的方式，还考查你头脑风暴的能力。你是否会像热锅上的蚂蚁，胡乱给出一些建议？抑或以合乎逻辑的、有条理的方式处理问题？希望是后者。

12.3 有个国际象棋游戏程序使用了方法：`boolean canMoveTo(int x, int y)`，这个方法是`Piece`类的一部分，可以判断某个棋子能否移动到位置(x, y)。请说明你会如何测试该方法。（第 83 页）

解法

这个问题主要涉及两大类测试：极限情况测试（确保有错误输入时程序不会崩溃）和一般情况测试。我们先从第一类测试开始。

测试类型 1：极限情况测试

确保程序会妥善处理错误或异常输入，这意味着要检查以下情况：

- 测试x和y为负数的情况；
- 测试x大于棋盘宽度的情况；
- 测试y大于棋盘高度的情况；
- 测试一个满是棋子的棋盘；
- 测试一个空或接近空的棋盘；
- 测试白子远多于黑子的情况；
- 测试黑子远多于白子的情况。

对于上面的错误情况，我们应该询问面试官，是要返回false还是抛出异常，然后有针对性地进行测试。

测试类型 2：一般情况测试

一般情况测试的涉及面要大得多。理想的做法是测试每一种可能的棋盘布局，但是棋局实在太多了。不过，我们还是可以合理地执行测试，尽量涵盖不同的棋局。

国际象棋一共有6种棋子，我们可以测试每一种棋子，在所有可能的方向上，向其他所有棋子移动的情况。大致如下面的代码所示：

```
对每一种棋子a：
    对其他每一种棋子b（6种及空白）
        对每一个方向d
            创建有a的棋盘
            将b放在方向d上
            试着移动——检查返回值
```

此题的关键在于，认识到我们不可能测试每一种可能的场景，即使有心也无力办到。相反，我们必须专注于最重要的部分。

12.4　不借助任何测试工具，该如何对网页进行负载测试？（第 83 页）

解法

负载测试（load test）不仅有助于定位Web应用性能的瓶颈，还能确定其最大连接数。同样地，它还能检查应用如何响应各种负载情况。

要进行负载测试，必须先确定对性能要求最高的场景，以及满足目标的性能衡量指标。一般来说，有待测量的对象包括：

- 响应时间；
- 吞吐量；
- 资源利用率；
- 系统所能承受的最大负载。

随后，我们设计各种测试模拟负载，细心测量上面的每一项。

若缺少正规的测试工具，我们可以自行打造。例如，可以创建成千上万的虚拟用户，模拟并发用户。我们会编写多线程的程序，新建成千上万个线程，每个线程扮演一个实际用户，载入待测页面。对于每个用户，可以利用程序来测量响应时间、数据I/O（输入/输出），等等。

之后，还要分析测试期间收集的数据结果，并与可接受的值进行比较。

12.5　如何测试一支笔？（第 83 页）

解法

这个问题很大程度上在于理解限制条件，并有条理、结构化地解决该问题。

为了理解有哪些限制条件，你应该抛出一系列疑问，针对某个问题了解“谁、什么、何地、何时、如何以及为什么”（只要与该问题相关，越多越好）。一个好的测试人员会在着手测试之前，先准确了解自己要测试的是什么。

为了说明上面这项技巧，我们来看看下面的模拟对话。

面试官：你会如何测试一支笔？

求职者：我想先了解一下这支笔。谁会使用这支笔？

面试官：可能是小孩。

求职者：嗯，有意思。他们会用这支笔做什么？写字、画画还是干别的？

面试官：画画。

求职者：好的，谢谢。画在哪里呢？纸张、布料还是墙壁上？

面试官：画在布料上。

求职者：那么，这支笔的笔头是什么样的？签字笔还是圆珠笔？要洗得掉的，还是洗不掉的？

面试官：要求洗得掉。

在问了很多问题之后，你可以得出如下结论。

求职者：好的，综上，我理解如下：这支笔主要面向5～10岁的小孩，为签字笔头，有红、绿、蓝、黑四色，用来画画。画在布料上并且要求洗得掉。我的理解对吗？

此时，求职者面对的问题与乍看上去的问题差异很大，这种情况并不少见。实际上，许多面试官会故意给一个看似再清楚不过的问题（谁不知道笔是什么呢！），其实是在考查你，看你能否发现这个问题与最初理解的有很大差别。他们相信用户也会这么做，但用户多半是无意的。

至此，你已经知道自己要测试的是什么，接下来该提出测试计划了。这里的关键是**结构**。

想想测试对象或问题会涉及哪些方面，并以此为基础展开测试。这个问题涉及以下几个方面。

- **事实核查**：核实这是一支签字笔，墨水颜色为要求的四种颜色之一。
- **预期用途**：绘制，这支笔在布料上画得出来吗？
- **预期用途**：水洗，画在布料上的墨迹洗得掉吗（哪怕已经过了一段时间）？是用热水、温水还是冷水才能洗掉？
- **安全性**：这支笔对小孩是否安全（无毒）？
- **非预期用途**：小孩还会怎么使用这支笔？他们可能在其他物体表面上涂鸦，因此还需检查他们的行为是否正确。他们还可能踩踏、乱扔这支笔，等等。你需要确认这支笔是否经受得住这些使用条件。

记住，对于任何测试问题，你都必须测试预期和非预期的场景。人们并不一定按照你预想的方式使用产品。

12.6 在一个分布式银行系统中，该如何测试一台ATM机？（第 83 页）

解法

对于这个问题，第一要务是厘清若干假设条件，请提出以下问题。

- 谁会使用ATM机？答案可能是“任何人”，或是“盲人”，或任意其他可能的答案。
- 他们会用ATM机来做什么？答案可能是“取款”、“转账”、“查询余额”，等等。
- 我们有什么工具来测试呢？我们可以查看代码吗？还是只能访问ATM机？

记住，好的测试人员会先确定自己要测试的是什么。

一旦了解系统是什么样的，我们就会想着将问题分解成可测试的子部分，包括：

- 登录；
- 取款；
- 存款；
- 查询余额；
- 转账。

我们可能要搭配使用手动和自动测试。

手动测试会检查上述步骤的每一个环节，确保涵盖所有错误情况（余额不足、新开账户、不存在的账户，等等）。

自动测试稍微复杂一点。我们会希望自动处理上述所有标准流程，还要找一些非常具体的问题，比如竞争条件。理想情况下，我们会设法建立一套有假帐户的封闭系统，以确保即使有人从不同地点快速取款和存款，他也不会多得不应得的钱，或者损失应得的钱。

最重要的是，我们必须优先考虑安全性和可靠性。客户的帐户无时无刻都要处于被保护的状

态，我们必须确保账目得到正确处理。没有人希望自己的钱不翼而飞。优秀的测试人员深谙整个系统里哪些事项是最重要的。

9.13　C 和 C++

13.1　用C++写个方法，打印输入文件的最后K行。（第 88 页）

解法

此题有一种蛮力法：先数出文件的行数（N），然后打印第$N-K$行到第N行。但是，这么做，文件要读两遍，会产生没必要的开销。我们需要一种解法，只读一遍文件就能打印最后K行。

我们可以使用一个数组，存放从文件读取到的所有K行和最后的K行。因此，这个数组起初包含的是$0 \sim K$行，然后是$1 \sim K+1$行，接着是$2 \sim K+2$行，依此类推。每次读取新的一行，就将数组中最早读入的那一行清掉。

不过，你可能会问，这么做是不是还要移动数组元素，进而引入很大的开销？不会，只要做法得当就不会。我们将使用循环式数组，而不必每次都移动数组元素。

使用循环式数组（circular array），每次读取新的一行，都会替换数组中最早读入的元素。我们会以专门的变量记录这个元素；每次加入新元素，该变量就要随之更新。

下面是循环式数组的例子：

```
步骤1（初始态）： array = {a, b, c, d, e, f}. p = 0
步骤2（插入g）：  array = {g, b, c, d, e, f}. p = 1
步骤3（插入h）：  array = {g, h, c, d, e, f}. p = 2
步骤4（插入i）：  array = {g, h, i, d, e, f}. p = 3
```

下面是该算法的实现代码。

```
void printLast10Lines(char* fileName) {
   const int K = 10;
   ifstream file (fileName);
   string L[K];
   int size = 0;

   /* 逐行读取文件，并存入循环式数组 */
   while (file.good()) {
     getline(file, L[size % K]);
     size++;
   }

   /* 计算循环式数组的开头和大小 */
   int start = size > K ? (size % K) : 0;
   int count = min(K, size);

   /* 根据读取顺序，打印数组元素 */
   for (int i = 0; i < count; i++) {
     cout << L[(start + i) % K] << endl;
   }
}
```

这种解法要求读取整个文件，不过，任意时刻都只会在内存里存放10行内容。

13.2 比较并对比散列表和STL map。散列表是如何实现的？如果输入的数据量不大，可以选用哪些数据结构替代散列表？（第 89 页）

解法

在散列表里，值的存放是通过将键传入散列函数实现的。值并不是以排序后的顺序存放。此外，散列表以键找出索引，进而找到存放值的地方，因此，插入或查找操作均摊后可以在$O(1)$时间内完成（假定该散列表很少发生碰撞冲突）。散列表还必须处理潜在的碰撞冲突，一般通过拉链法（chaining）解决，也即创建一个链表来存放值，这些值的键都映射到同一个索引。

STL map的做法是根据键，将键值对插入二叉查找树。不需要处理冲突，因为树是平衡的，插入和查找操作的时间肯定为$O(\log N)$。

散列表是如何实现的？

传统上，散列表都是用元素为链表的数组实现的。想要插入键值对时，先用散列函数将键映射为数组索引，随后，将值插入那个索引位置对应的链表。

注意，在数组的特定索引位置的链表中，各个元素的键并不相同，这些值的`hashFunction(key)`才是相同的。因此，为了取回某个键对应的值，每个结点都必须存放键和值。

总而言之，散列表会以链表数组的形式实现，链表中每个结点都会存放两块数据：值和原先的键。此外，我们还要注意以下设计准则。

(1) 我们希望使用一个优良的散列函数，确保能将键均匀分散开来。若分散不均匀，就会发生大量碰撞冲突，查找元素的速度也会变慢。

(2) 不论散列函数选的多好，还是会出现碰撞冲突，因此需要一种碰撞处理方法。通常，我们会采用拉链法，也就是通过链表来处理，但这并不是唯一的做法。

(3) 我们可能还希望设法根据容量动态扩大或缩小散列表的大小。例如，当元素数量和散列表大小之比超过一定阈值时，可能会希望扩大散列表的大小。这意味着要新建一个散列表，并将旧的散列表条目转移到新的散列表中。因为这种操作的开销非常大，所以我们要谨慎些，切不可频繁操作。

如果输入的数据量不大，可以选用哪些数据结构替代散列表？

你可以使用STL map或二叉树。尽管两者的插入操作需要$O(\log(n))$的时间，但若是输入数据量够小，这点时间就可以忽略不计。

13.3 C++虚函数的工作原理是什么？（第 89 页）

解法

虚函数（virtual function）需要虚函数表（vtable，Virtual Table）才能实现。如果一个类有函数声明成虚拟的，就会生成一个vtable，存放这个类的虚函数地址。此外，编译器还会在类里加入隐藏的`vptr`变量。若子类没有覆写虚函数，该子类的vtable就会存放父类的函数地址。调用这

个虚函数时，就会通过vtable解析函数的地址。在C++里，动态绑定（dynamic binding）就是通过vtable机制实现的。

由此，将子类对象赋值给基类指针时，vptr变量就会指向子类的vtable。这样一来，就能确保继承关系最末端的子类虚函数会被调用到。

请考虑以下代码。

```
class Shape {
  public:
   int edge_length;
   virtual int circumference () {
     cout << “Circumference of Base Class\n”;
     return 0;
   }
};
class Triangle: public Shape {
  public:
   int circumference () {
     cout<< “Circumference of Triangle Class\n”;
     return 3 * edge_length;
   }
};
void main() {
   Shape * x = new Shape();
   x->circumference(); // “Circumference of Base Class”
   Shape *y = new Triangle();
   y->circumference(); // “Circumference of Triangle Class”
}
```

在上面的代码中，circumference是Shape类的虚函数，因此在所有继承Shape类的子类（Triangle等）里都为虚函数。在C++里，非虚函数的调用是在编译期通过静态绑定确定的，而虚函数的调用则是在运行期通过动态绑定确定的。

13.4　深拷贝和浅拷贝之间有何区别？请说明两者的用法。（第 89 页）

解法

浅拷贝会将对象所有成员的值拷贝到另一个对象里。除了拷贝所有成员的值，深拷贝还会进一步拷贝所有指针对象。

下面是浅拷贝和深拷贝的例子。

```
struct Test {
   char * ptr;
};

void shallow_copy(Test & src, Test & dest) {
   dest.ptr = src.ptr;
}

void deep_copy(Test & src, Test & dest) {
   dest.ptr = (char *)malloc(strlen(src.ptr) + 1);
```

```
    strcpy(dest.ptr, src.ptr);
}
```

注意，shallow_copy可能会导致大量编程运行时错误，尤其是在对象创建和销毁时。使用浅拷贝时，必须非常小心，只有当开发人员真正知道自己在做些什么时方可选用浅拷贝。多数情况下，使用浅拷贝是为了传递一块复杂结构的信息，但又不想真的复制一份数据。使用浅拷贝时，销毁对象必须非常小心。

在实际开发中，浅拷贝很少使用。大部分情况都应该使用深拷贝，特别是当需要拷贝的结构很小时。

13.5 C语言的关键字“volatile”有何作用？（第 89 页）

解法

关键字volatile的作用是指示编译器，即使代码不对变量做任何改动，该变量的值仍可能会被外界修改。操作系统、硬件或其他线程都有可能修改该变量。该变量的值有可能遭受意料之外的修改，因此，每一次使用时，编译器都会重新从内存中获取这个值。

volatile（易变）的整数可由下面的语句声明：

```
int volatile x;
volatile int x;
```

要声明指向volatile整数的指针，可以这么做：

```
volatile int * x;
int volatile * x;
```

指向非volatile数据的volatile指针很少见，但也是可行的：

```
int * volatile x;
```

如若声明指向一块volatile内存的volatile指针变量（指针本身与地址所指的内存都是volatile），做法如下：

```
int volatile * volatile x;
```

volatile变量不会被优化掉，这非常有用。设想有下面这个函数：

```
int opt = 1;
void Fn(void) {
   start:
     if (opt == 1) goto start;
     else break;
}
```

乍一看，上面的代码好像会进入无限循环，编译器可能会将这段代码优化成：

```
void Fn(void) {
   start:
     int opt = 1;
     if (true)
     goto start;
}
```

这样就变成了无限循环。然后，外部操作可能会将0写入变量opt的位置，从而终止循环。

为了防止编译器执行这类优化，我们需要设法通知编译器，系统其他部分可能会修改这个变量。具体做法就是使用volatile关键字，如下所示。

```
volatile int opt = 1;
void Fn(void) {
   start:
     if (opt == 1) goto start;
     else break;
}
```

volatile变量在多线程程序里也很有用，对于全局变量，任意线程都可能修改这些共享的变量。我们可能不希望编译器对这些变量进行优化。

13.6　基类的析构函数为何要声明为virtual？（第 89 页）

解法

让我们先想想为何会有虚函数，假设有如下代码：

```
class Foo {
 public:
   void f();
};

class Bar : public Foo {
 public:
   void f();
}

Foo * p = new Bar();
p->f();
```

调用p->f()最后将会调用Foo::f()，这是因为p是指向Foo的指针，而f()不是虚拟的。

为确保p->f()会调用继承关系最末端的子类的f()实现，我们需要将f()声明为虚函数。

现在，回到前面的析构函数。析构函数用于释放内存和资源。Foo的析构函数若不是虚拟的，那么，即使p实际上是Bar类型的，还是会调用Foo的析构函数。

这就是为何要将析构函数声明为虚拟的原因——确保正确调用继承关系最末端的子类的析构函数。

13.7 编写方法，传入参数为指向Node结构的指针，返回传入数据结构的完整拷贝。其中，Node数据结构含有两个指向其他Node的指针。（第 89 页）

解法

下面的算法将记录一份映射关系，从原先结构中的结点地址对应到新结构中相应的结点。利用该映射关系，在这个结构的深度优先遍历中，就能判断某个结点是不是复制过了。遍历时通常会标记访问过的结点，标记可以有多种形式，不一定要存放在结点里。

综上，可以得到一个简单的递归算法：

```
typedef map<Node*, Node*> NodeMap;

Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
   if(cur == NULL) {
     return NULL;
   }

   NodeMap::iterator i = nodeMap.find(cur);
   if (i != nodeMap.end()) {
     // 已访问过这里，返回拷贝
     return i->second;
   }

   Node * node = new Node;
   nodeMap[cur] = node; // 在遍历链接之前，建立映射关系
   node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
   node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
   return node;
}

Node * copy_structure(Node * root) {
   NodeMap nodeMap; // 需要一个空的map
   return copy_recursive(root, nodeMap);
}
```

13.8 编写一个智能指针类。智能指针是一种数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录SmartPointer<T*>对象的引用计数，一旦T类型对象的引用计数为零，就会释放该对象。（第 89 页）

解法

智能指针跟普通指针一样，但它借由自动化内存管理保证了安全性，避免了诸如悬挂指针、内存泄漏和分配失败等问题。智能指针必须为给定对象的所有引用维护单一引用计数。

第一次看到这类问题，可能会觉得太难而不知所措，特别是当你并非C++专家时。此题有个解决之道，分两步走：(1) 以伪码勾勒出做法；(2) 实现具体代码。

按照这种做法，我们需要一个引用计数变量，每新增一个对象的引用，该变量会加一，移除一个引用则减一。实现代码与下面的伪码类似：

```
template <class T> class SmartPointer {
   /* 智能指针类需要指向对象本身及引用计数两者
    * 的指针。这些都必须是指针，而不是真实的对象
    * 或引用计数值，因为智能指针的目的就在于，
    * 可以跨多个指向某一对象的智能指针，来追踪
    * 同一个引用计数 */
   T * obj;
   unsigned * ref_count;
}
```

这个类还需要若干构造函数和一个析构函数，下面先加上这些函数。

```
SmartPointer(T * object) {
```

```
  /* 想要设定T * obj的值，并将引用计数
   * 设为1 */
}

SmartPointer(SmartPointer<T>& sptr) {
  /* 这个构造函数会新建一个指向已有对象的
   * 智能指针。我们需要先设定obj和ref_count，
   * 设为指向sptr的obj和ref_count。然后，
   * 因为我们新建了一个obj的引用，所以需要
   * 增加ref_count */
}

~SmartPointer(SmartPointer<T> sptr) {
  /* 销毁该对象的引用，减少ref_count的值。
   * 若ref_count为0，则释放为存放整数而申请的内存，
   * 并销毁对象 */
}
```

还有一种方式也可以创建引用：将一个SmartPointer赋值给另一个。处理这种情况需要覆写=操作符，不过这里先略述一二。

```
onSetEquals(SmartPointer<T> ptr1, SmartPointer<T> ptr2) {
  /* 若ptr1已有值，减小其引用计数。然后，
   * 复制指向obj和ref_count的指针。最后，
   * 因为创建了新引用，所以需要增加
   * ref_count的值 */
}
```

即使尚未填入复杂的C++语法，仅仅把做法大致描绘出来，意义已经很重大了。接下来，要完成所有代码，只需填补好细节即可。

```
template <class T> class SmartPointer {
 public:
  SmartPointer(T * ptr) {
    ref = ptr;
    ref_count = (unsigned*)malloc(sizeof(unsigned));
    *ref_count = 1;
  }

  SmartPointer(SmartPointer<T> & sptr) {
    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
  }

  /* 覆写=运算符，这样才能将一个旧的
   * 智能指针赋值给另一指针，旧的引用
   * 计数减一，新的智能指针的引用计数
   * 则加一 */
  SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
    if (this == &sptr) return *this;

    /* 若已赋值为某个对象，则移除引用 */
    if (*ref_count > 0) {
```

```
      remove();
    }

    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
    return *this;
  }

  ~SmartPointer() {
    remove(); // 移除一个对象引用
  }

  T getValue() {
    return *ref;
  }

 protected:
  void remove() {
    --(*ref_count);
    if (*ref_count == 0) {
      delete ref;
      free(ref_count);
      ref = NULL;
      ref_count = NULL;
    }
  }

  T * ref;
  unsigned * ref_count;
};
```

此题的代码复杂难懂，错漏在所难免，面试官也不会强求代码写得完美无缺。

13.9　编写支持对齐分配的malloc和free函数，分配内存时，malloc函数返回的地址必须能被 2 的*n*次方整除。（第 89 页）

解法

一般来说，使用`malloc`，我们控制不了分配的内存会在堆里哪个位置。我们只会得到一个指向内存块的指针，指针的起始地址不定。

要克服这些限制条件，我们必须申请足够大的内存，要大到可以返回可被指定数值整除的内存地址。

假设需要一个100字节的内存块，我们希望它的起始地址为16的倍数。需要额外分配多少内存才够用呢？我们需要额外分配15字节。有了这15字节，加上紧随其后的100字节，就能得到可被16整除的内存地址，以及100字节的可用空间。

具体做法大致如下：

```
void* aligned_malloc(size_t required_bytes, size_t alignment) {
  int offset = alignment - 1;
```

```
3     void* p = (void*) malloc(required_bytes + offset);
4     void* q = (void*) (((size_t)(p) + offset) & ~(alignment - 1));
5     return q;
6  }
```

第4行有点难懂，解释如下。假设alignment为16。很显然，在前16字节的某个位置，肯定有个内存地址可被16整除。执行(p1 + 16) & 11..10000，可将q往后移到可被16整除的内存地址。并地址末4位和0000执行位与操作，以确保新的值可被16整除。

这种解法近乎无可挑剔，只是有个大问题：如何释放这块内存？

在上面的代码中，我们额外分配了15字节，在释放“真正的”内存时，必须释放这块额外内存。

为了释放整个内存块，我们可以将它的起始地址存放在这块“额外”内存中。我们会在紧邻地址对齐的内存块之前，存放这个地址。当然，这意味着我们现在需要更多的额外内存，以确保有足够的空间存放这个起始地址。

特别是，对于按alignment字节数对齐，我们需要额外分配alignment - 1 + sizeof(void*)字节。

下面是该做法的实现代码。

```
void* aligned_malloc(size_t required_bytes, size_t alignment) {
   void* p1;  // 原先的内存块
   void** p2; // 对齐后的内存块
   int offset = alignment - 1 + sizeof(void*);
   if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
     return NULL;
   }
   p2 = (void**)(((size_t)(p1) + offset) & ~(alignment - 1));
   p2[-1] = p1;
   return p2;
}

void aligned_free(void *p2) {
   /* 为了一致性，这里也仿照aligned_malloc函数取名 */
   void* p1 = ((void**)p2)[-1];
   free(p1);
}
```

下面看看aligned_free是怎么运作的，该函数有个传入参数为p2（与aligned_malloc里的p2是相同的）。很显然，p1的值（指向完整内存块的开头）就存放在p2的前面。

如果我们把p2看作void**（或者void *的数组），就可以按索引- 1取得p1。然后，释放p1就可释放整块内存。

13.10 用C编写一个my2DAlloc函数，可分配二维数组。将malloc函数的调用次数降到最少，并确保可通过arr[i][j]访问该内存。（第 89 页）

解法

大家可能都知道，二维数组本质上就是数组的数组。既然可以用指针访问数组，就可以用双

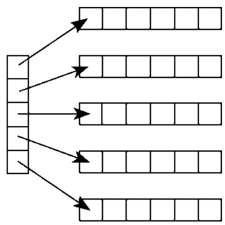
重指针来创建二维数组。

基本思路是先创建一个一维指针数组。然后，为每个数组索引，再新建一个一维数组。这样就能得到一个二维数组，可通过数组索引访问。

下面是该做法的实现代码。

```
int** my2DAlloc(int rows, int cols) {
   int** rowptr;
   int i;
   rowptr = (int**) malloc(rows * sizeof(int*));
   for (i = 0; i < rows; i++)  {
     rowptr[i] = (int*) malloc(cols * sizeof(int));
   }
   return rowptr;
}
```

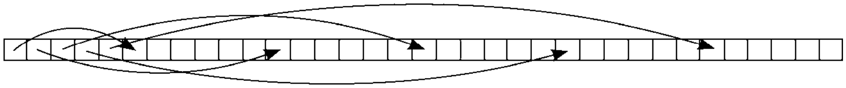
仔细观察上面的代码，注意我们是怎样让rowptr根据索引指向具体位置的。下图显示了内存是怎么分配的。



释放这些内存不能直接对rowptr调用free。我们要确保不仅释放掉第一次malloc调用分配的内存，还要释放后续每次malloc调用分配的内存。

```
void my2DDealloc(int** rowptr, int rows) {
   for (i = 0; i < rows; i++) {
     free(rowptr[i]);
   }
   free(rowptr);
}
```

我们还可以分配一大块连续的内存，这样就不必分配很多个内存块（每一行一块，外加一块内存，存放每一行的首地址）。举个例子，对于五行六列的二维数组，这种做法的效果如下图所示。



看到这样的二维数组似乎有点奇怪，注意，它与前一张图并没什么不同。唯一区别是现在是一大块连续的内存，因此，此例中前五个元素指向同一块内存的其他位置。

下面是这种做法的具体实现。

```
int** my2DAlloc(int rows, int cols) {
```

```
2     int i;
3     int header = rows * sizeof(int*);
4     int data = rows * cols * sizeof(int);
5     int** rowptr = (int**)malloc(header + data);
6     if (rowptr == NULL) {
7       return NULL;
8     }
9
10    int* buf = (int*) (rowptr + rows);
11    for (i = 0; i < rows; i++) {
12      rowptr[i] = buf + i * cols;
13    }
14    return rowptr;
15 }
```

注意，仔细观察第11 ~ 13行代码的具体实现。假设该二维数组有五行，每行六列，则array[0]会指向array[5]，array[1]指向array[11]，依此类推。

随后，当我们真正调用array[1][3]时，计算机会查找array[1]，这是个指针，指向内存的另一个地方，其实就是指向array[5]的指针。这个元素会被视为一个数组，然后取出它的第3个元素（索引从0开始）。

用这种方法构建数组只需调用一次malloc，另外还有个好处，就是清除数组时也只需调一次free，而不必专门写个函数释放其余的内存块。

9.14　Java

14.1　从继承的角度来看，将构造函数声明为私有会有何作用？（第 93 页）

解法

将构造函数声明为私有（private），可确保类以外的地方都不能直接实例化这个类。在这种情况下，要创建这个类的实例，唯一的办法是提供一个公共静态方法，就像工厂方法模式（Factory Method Pattern）那样。

此外，由于构造函数是私有的，因此这个类也不能被继承。

14.2　在Java中，若在try-catch-finally的try语句块中插入return语句，finally语句块是否还会执行？（第 93 页）

解法

是的，它会执行。当退出try语句块时，finally语句块将会执行。即使我们试图从try语句块里跳出（通过return语句、continue语句、break语句或任意异常），finally语句块仍将得以执行。

注意，有些情况下finally语句块将不会执行，比如：

- 如果虚拟机在try/catch语句块执行期间退出；
- 如果执行try/catch语句块的线程被杀死终止了。

14.3 final、finally和finalize之间有何差异？（第 93 页）

解法

尽管名字相像、发音类似，final、finally和finalize的功能截然不同。非常笼统地说，final用于控制变量、方法或类是否“可更改”。finally关键字用在try/catch语句块中，以确保一段代码一定会被执行。一旦垃圾收集器确定没有任何引用指向某个对象，就会在销毁该对象之前调用finalize()方法。

下面是关于这几个关键字和方法的更多细节。

1. final

上下文不同，final语句含义有别。

- 应用于基本类型（primitive）变量时：该变量的值无法更改。
- 应用于引用（reference）变量时：该引用变量不能指向堆上的任何其他对象。
- 应用于方法时：该方法不允许重写。
- 应用于类时：该类不能派生子类。

2. finally

在try块或catch块之后，可以选择加一个finally语句块。finally语句块里的语句一定会被执行（除非Java虚拟机在执行try语句块期间退出）。我们会在finally语句块里编写资源回收和清理的代码。

3. finalize()

当垃圾收集器确定再无任何引用指向某个对象实例时，就会在销毁该对象之前调用finalize()方法，一般用于清理资源，比如关闭文件。

14.4 C++模板和Java泛型之间有何不同？（第 93 页）

解法

许多程序员都认为模板（template）和泛型（generic）这两个概念是等价的，因为两者都允许你按照List<String>的样式编写代码。不过，各种语言是怎么实现该功能的，以及为什么这么做，却千差万别。

Java泛型的实现植根于“类型消除”这一概念。当源代码被转换成Java虚拟机字节码时，这种技术会消除参数化类型。

例如，假设有以下Java代码：

```
Vector<String>  vector = new Vector<String>();
vector.add(new String(“hello”));
String  str = vector.get(0);
```

编译时，上面的代码会被改写为：

```
Vector vector = new Vector();
vector.add(new String(“hello”));
```

```
String str = (String) vector.get(0);
```

有了Java泛型，我们可以做的事情也并没有真正改变多少；它只是让代码变得漂亮些。鉴于此，Java泛型有时也被称为“语法糖”。

这点跟C++的模板截然不同。在C++中，模板本质上就是一套宏指令集，只是换了个名头，编译器会针对每种类型创建一份模板代码的副本。有项证据可以证明这一点：`MyClass<Foo>`不会与`MyClass<Bar>`共享静态变量。然而，两个`MyClass<Foo>`实例则会共享静态变量。

看了下面的代码，应该会更清楚一点：

```
/*** MyClass.h ***/
template<class T> class MyClass {
 public:
   static int val;
   MyClass(int v) { val = v; }
};

/*** MyClass.cpp ***/
template<typename T>
int MyClass<T>::bar;

template class MyClass<Foo>;
template class MyClass<Bar>;

/*** main.cpp ***/
MyClass<Foo> * foo1 = new MyClass<Foo>(10);
MyClass<Foo> * foo2 = new MyClass<Foo>(15);
MyClass<Bar> * bar1 = new MyClass<Bar>(20);
MyClass<Bar> * bar2 = new MyClass<Bar>(35);

int f1 = foo1->val; // 等于15
int f2 = foo2->val; // 等于15
int b1 = bar1->val; // 等于35
int b2 = bar2->val; // 等于35
```

在Java中，`MyClass`类的静态变量会由所有`MyClass`实例共享，不论类型参数相同与否。

由于架构设计上的差异，Java泛型和C++模板还有如下很多不同点。

- C++模板可以使用`int`等基本数据类型。Java则不行，必须转而使用`Integer`。
- 在Java中，可以将模板的类型参数限定为某种特定类型。例如，你可能会使用泛型实现`CardDeck`，并规定类型参数必须扩展自`CardGame`。
- 在C++中，类型参数可以实例化，但Java不支持。
- 在Java中，类型参数（即`MyClass<Foo>`中的`Foo`）不能用于静态方法和变量，因为它们会被`MyClass<Foo>`和`MyClass<Bar>`所共享。在C++中，这些类都是不同的，因此类型参数可以用于静态方法和静态变量。
- 在Java中，不管类型参数是什么，`MyClass`的所有实例都是同一类型。类型参数会在运行时被抹去。在C++中，参数类型不同，实例类型也不同。

记住，Java泛型和C++模板，虽然在很多方面看起来都一样，但实则大不相同。

14.5 Java中的对象反射是什么？它有什么用？（第 93 页）

解法

对象反射（Object Reflection）是Java的一项特性，提供了获取Java类和对象的反射信息的方法，可执行如下操作。

(1) 运行时取得类的方法和字段的相关信息。

(2) 创建某个类的新实例。

(3) 通过取得字段引用直接获取和设置对象字段，不管访问修饰符为何。

下面这段代码为对象反射的示例。

```
/* 参数 */
Object[] doubleArgs = new Object[] { 4.2, 3.9 };

/* 取得类 */
Class rectangleDefinition = Class.forName(“MyProj.Rectangle”);

/* 等同于：Rectangle rectangle = new Rectangle(4.2, 3.9); */
Class[] doubleArgsClass = new Class[] {double.class, double.class};
Constructor doubleArgsConstructor =
   rectangleDefinition.getConstructor(doubleArgsClass);
Rectangle rectangle =
   (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);

/* 等同于：Double area = rectangle.area(); */
Method m = rectangleDefinition.getDeclaredMethod(“area”);
Double area = (Double) m.invoke(rectangle);
```

这段代码等同于：

```
Rectangle rectangle = new Rectangle(4.2, 3.9);
Double area = rectangle.area();
```

对象反射有什么用？

当然，从上面的例子来看，对象反射似乎没什么用，不过在特定情况下反射可能非常有用。

对象反射之所以有用，主要体现在以下3个方面。

(1) 有助于观察或操纵应用程序的运行时行为。

(2) 有助于调试或测试程序，因为我们可以直接访问方法、构造函数和成员字段。

(3) 即使事前不知道某个方法，我们也可以通过名字调用该方法。例如，让用户传入类名、构造函数的参数和方法名。然后，我们就可以使用该信息来创建对象，并调用方法。如果没有反射的话，即使可以做到，也需要一系列复杂的if语句。

14.6 实现`CircularArray`类，支持类似数组的数据结构，这些数据结构可以高效地进行旋转。该类应该使用泛型，并通过标准的`for (Obj o : circularArray)`语法支持迭代操作。（第 93 页）

解法

此题实际上有两部分。首先，我们需要实现`CircularArray`类。其次，需要支持迭代。下面

将分别加以说明。

实现 CircularArray 类

实现`CircularArray`类的方式之一是每次调用`rotate(int shiftRight)`时都要移动元素。当然，这么做效率低下。

反之，我们可以只创建一个成员变量`head`，指向概念上应被视作循环数组开头的元素。我们不必四处移动数组元素，只需通过`shiftRight`增加`head`的值。

下面是该做法的实现代码。

```
public class CircularArray<T> {
   private T[] items;
   private int head = 0;

   public CircularArray(int size) {
     items = (T[]) new Object[size];
   }

   private int convert(int index) {
     if (index < 0) {
        index += items.length;
     }
     return (head + index) % items.length;
   }

   public void rotate(int shiftRight) {
     head = convert(shiftRight);
   }

   public T get(int i) {
     if (i < 0 || i >= items.length) {
        throw new java.lang.IndexOutOfBoundsException("...");
     }
     return items[convert(i)];
   }

   public void set(int i, T item) {
     items[convert(i)] = item;
   }
}
```

其中有几个地方很容易出错，比如：

- 我们无法创建泛型的数组。相反，我们必须将数组转型或者将`items`类型定义为`List<T>`。为了简单起见，这里选用了前一种做法。
- 执行`negValue % posVal`（负值 % 正值）时，%操作符会返回负值。举个例子，`-8 % 3`的结果为`-2`。这跟数学家定义的取模函数不同，我们必须将负数索引加上`items.length`，以得到正确的正数值。
- 无论何时都必须确保将原索引转成旋转后的索引。为此，我们实现了`convert`函数供其他函数使用。即使`rotate`函数也会使用`convert`。这是一个很好的代码复用的范例。

现在，我们明确了CircularArray的基本代码，接下来可以专注于迭代器的实现。

实现迭代器（Iterator）接口

此题的第二部分要求我们实现CircularArray类之后，可以这么写代码：

```
CircularArray<String> array = ...
for (String s : array) { ... }
```

要做到这一点，就必须实现Iterator接口。

为实现Iterator接口，我们需要做到以下两点。

- 修改CircularArray<T>定义，添加implements Iterable<T>，同时还要在CircularArray<T>里加入iterator()方法。
- 创建实现Iterator<T>的CircularArrayIterator<T>，同时，还要在CircularArrayIterator里实现方法hasNext()、next()和remove()。

完成上述工作后，for循环就会如魔法般地发挥作用。

为节省篇幅，以下代码中与之前CircularArray实现相同的部分已删除。

```
public class CircularArray<T> implements Iterable<T> {
   ...
   public Iterator<T> iterator() {
     return new CircularArrayIterator<T>(this);
   }

   private class CircularArrayIterator<TI> implements Iterator<TI>{
     /* _current反映从旋转后的开头算起的偏移值，
      * 而不是从原始数组的开头算起 */
     private int _current = -1;
     private TI[] _items;

     public CircularArrayIterator(CircularArray<TI> array){
        _items = array.items;
     }

     @Override
     public boolean hasNext() {
        return _current < items.length - 1;
     }

     @Override
     public TI next() {
        _current++;
        TI item = (TI) _items[convert(_current)];
        return item;
     }

     @Override
     public void remove() {
        throw new UnsupportedOperationException("...");
     }
   }
}
```

注意，在上面的代码中，当for循环第一次迭代时，会调用hasNext()，然后是next()。务必确保你的实现会返回正确的值。

在面试中碰到类似题目时，很有可能想不起来需要调用哪些方法和接口。在这种情况下，你还是应该竭尽所能地解题。如果你能推导出可能需要用到哪些方法，光这样就能向面试官展现出你具备的能力。

9.15 数据库

问题1～3用到以下数据库模式：

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar
BuildingID	int	BuildingName	varchar		
		Address	varchar		

Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

注意，每套公寓可能有多位承租人，而每位承租人可能租住多套公寓。每套公寓隶属于一栋大楼，而每栋大楼属于一个综合体。

15.1 编写SQL查询，列出租住不止一套公寓的承租人。（第 97 页）

解法

要解决此题，我们可以使用HAVING和GROUP BY子句，然后将Tenants以INNER JOIN连接起来。

```
SELECT TenantName
FROM Tenants
INNER JOIN
   (SELECT TenantID
    FROM AptTenants
    GROUP BY TenantID
    HAVING count(*) > 1) C
ON Tenants.TenantID = C.TenantID
```

在面试或现实生活中，每当编写GROUP BY子句时，务必确保SELECT子句里的任何东西，要么是聚集函数，要么就是包含在GROUP BY子句里。

15.2 编写SQL查询，列出所有建筑物，并取得状态为“Open”的申请数量（Requests表中Status为Open的条目）。（第 97 页）

解法

此题直接将Requests和Apartments连接起来，就能列出建筑物ID，并取得Open申请的数量。取得这份列表后，再将它与Buildings表进行连接。

```
SELECT BuildingName, ISNULL(Count, 0) as 'Count'
FROM Buildings
LEFT JOIN
   (SELECT Apartments.BuildingID, count(*) as ‘Count’
   FROM Requests INNER JOIN Apartments
   ON Requests.AptID = Apartments.AptID
   WHERE Requests.Status = ‘Open’
   GROUP BY Apartments.BuildingID) ReqCounts
ON ReqCounts.BuildingID = Buildings.BuildingID
```

诸如这种有子查询的查询，务必要经过全面测试，手写时尤当如此。最好先测试查询的内层，然后再测试外层部分。

15.3 11 号建筑物正在进行大翻修。编写SQL查询，关闭这栋建筑物里所有公寓的入住申请。（第 97 页）

解法

跟SELECT查询一样，UPDATE查询也可以有WHERE子句。要实现这个查询，我们会获取11号建筑物里所有公寓的ID，然后从这些公寓取得入住申请列表。

```
UPDATE Requests
SET Status = ‘Closed’
WHERE AptID IN
   (SELECT AptID
    FROM Apartments
    WHERE BuildingID = 11)
```

15.4 连接有哪些不同类型？请说明这些类型之间的差异，以及为何在某些情形下，某种连接会比较好。（第 97 页）

解法

JOIN用于合并两个表的结果。要执行JOIN操作，每个表里至少要有一个字段，可用来配对另一个表里的记录。连接的类型规定了哪些记录会进入合并结果集。

下面以两张表为例：一张表列出常规饮料，另一张表是无卡路里饮料。每张表有两个字段：饮料名称（name）和产品编号（code）。编号（code）字段用来配对记录。

常规饮料：

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA
Pepsi	PEPSI

无卡路里饮料：

Name	Code
Diet Coca-Cola	COCACOLA
Fresca	FRESCA
Diet Pepsi	PEPSI
Pepsi Light	PEPSI
Purified Water	Water

欲将Beverage与Calorie-Free Beverages连接起来，我们可以有多种选择，说明如下。

- INNER JOIN：结果集只含有配对成功的数据。在这个例子里，我们会得到三条记录：一条包含COCACOLA编号，两条包含PEPSI编号。
- OUTER JOIN：OUTER JOIN一定会包含INNER JOIN的结果，不过它也可能包含一些在其他表里没有配对的记录。OUTER JOIN还可分为以下几种子类型。
  - LEFT OUTER JOIN或简称LEFT JOIN：结果会包含左表的所有记录。如果右表中找不到配对成功的记录，则相应字段的值为NULL。在这个例子里，我们会得到四条记录。除了INNER JOIN的结果，还会列出BUDWEISER，因为它位于左表中。
  - RIGHT OUTER JOIN或简称RIGHT JOIN：这种连接刚好与LEFT JOIN相反。它会返回包括右表的所有记录；左表缺失的字段为NULL。注意，如果有两张表A和B，那么，可以认为语句A LEFT JOIN B与B RIGHT JOIN A等价。在上面的例子里，我们会得到五条记录。除了INNER JOIN结果，还会有FRESCA和WATER两条记录。
  - FULL OUTER JOIN：这种连接会合并LEFT和RIGHT JOIN的结果。不论另一个表里有无配对记录，这两个表的所有记录都会放进结果集中。如果找不到配对记录，则对应的结果字段的值为NULL。在这个例子里，我们会得到六条记录。

15.5　什么是反规范化？请说明优缺点。（第 97 页）

解法

反规范化（denormalization）是一种数据库优化技术，在一个或多个表中加入冗余数据。在使用关系型数据库中，反规范化可帮助我们避免开销很大的表连接操作。

相比之下，在传统的规范化数据库中，我们会将数据存放在不同的逻辑表里，试图将冗余数据减到最少，力争做到在数据库中每块数据只有一份副本。

例如，在规范化数据库中，我们可能会有Courses表和Teachers表。在Courses里，每个条

目都会储存课程（Course）的teacherID，但不存储teacherName。如欲获取所有课程（Courses）对应的教师（Teacher）姓名，只需对这两个表进行连接。

就某些方面来看，这么做很不错。如有教师更改名字，我们只需更新一个地方的名字。

不过，这么做的缺点在于，如果表很大，就需要花费过长时间对这些表执行连接操作。

而反规范化则可以达成一定的平衡。在反规范化时，我们确定自己可以接受一定的冗余，并在更新数据库时要多做些工作，从而减少连接操作，保证较高的效率。

反规范化的缺点	反规范化的优点
更新和插入操作开销更大	连接操作较少，因此检索数据更快
反规范化会使更新和插入代码更难写	需要查找的表较少，因此检索查询比较简单（因而也不容易出错）
数据可能不一致。哪一块数据才是“正确”的呢？	
数据存在冗余，需要更大的存储空间	

在注重可扩展性的系统中，比如大型科技公司，几乎一定会兼用规范化和反规范化数据库的各种要素。

15.6　有个数据库，里面有公司（companies）、人（people）和专业人员（professionals，为公司工作），请绘制实体关系图。（第 97 页）

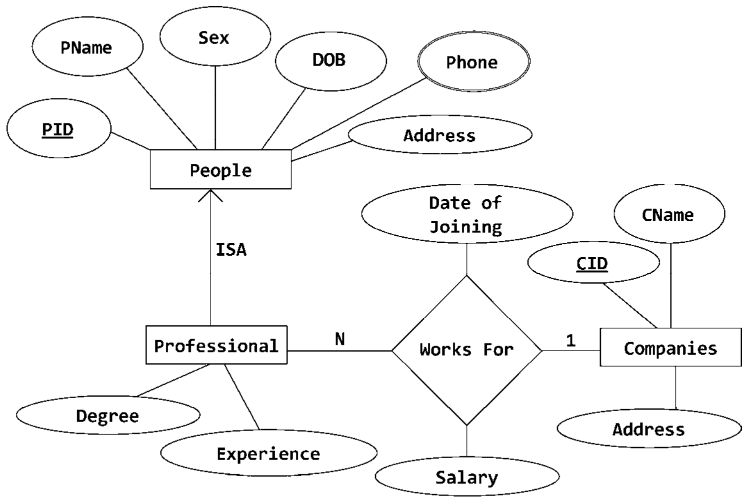
解法

在公司（Companies）上班的人（People）称作专业人员（Professional）。因此，People和Professional之间是ISA（“is a”）关系（或者说Professional派生自People）。

除了从People派生的属性，Professional还有一些附加信息，包括学历（degree）和工作经验（experience）等。

每位Professional同一时间只能为一家Company工作，而Companies则可以同时雇佣多位Professional，因此Professional和Companies之间是多对一的关系。“Works For”关系可以存放员工的入职时间和薪资等属性。这些属性只有在将Professional与Company相关联时才会定义。

一个People可能拥有多个电话号码，所以Phone是个多值属性。



15.7　给定一个储存有学生成绩的简单数据库。设计这个数据库的大概样子，并编写SQL查询，返回优等生名单（排名前 10%），以平均分排序。（第 97 页）

解法

在一个简单的数据库中，最起码会有三个对象：Students（学生）、Courses（课程）和CourseEnrollment（选修课程）。Students至少会有学生姓名、学号（ID），还可能包含其他个人信息。Courses会包含课程名和代号，或许还有课程说明、教授和其他信息。CourseEnrollment会将Students和Courses配对起来，还会含有Grade[①]字段。

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	float
Term	int

① 原文为CourseGrade。——译者注

要是加上教授的资料、学分费用信息和其他数据，这个数据库就会变得相当复杂。

使用微软SQL Server里的`TOP ... PERCENT`函数，我们可以先尝试如下（错误的）查询：

```
/* 错误代码 */
SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
   CourseEnrollment.StudentID
FROM CourseEnrollment
GROUP BY CourseEnrollment.StudentID
ORDER BY AVG(CourseEnrollment.Grade)
```

以上代码的问题在于，它只会如实返回按GPA排序后的前10%行记录。设想这样一个场景：有100名学生，排名前15的学生的GPA都是4.0。上面的函数只会返回其中10名学生，与我们的要求不符。在得分相同的情况下，我们希望计入得分前10%的学生，即使优等生名单的人数超过班级总人数的10%.

为纠正这个问题，我们可以建立类似的查询，不过首先要取得筛选优等生的GPA基准。

```
DECLARE @GPACutOff float;
SET @GPACutOff = (SELECT min(GPA) as 'GPAMin'
     FROM (
        SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
        FROM CourseEnrollment
        GROUP BY CourseEnrollment.StudentID
                       ORDER BY GPA desc)
     Grades);
```

接着，定义好`@GPACutOff`后，要筛选最低拥有该GPA的学生，就相当容易了。

```
SELECT StudentName, GPA
FROM (
   SELECT AVG(CourseEnrollment.Grade) AS GPA,
          CourseEnrollment.StudentID
   FROM CourseEnrollment
   GROUP BY CourseEnrollment.StudentID
   HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

对于你所做出的隐含假设条件，必须非常小心。仔细查看上面的数据库描述，你会发现哪些可能是不正确的假设？其中之一是每门课程只能由一位教授来教。而在某些学校，一门课程可能会由多位教授来教。

不过，你还是需要做出一些假设，要不然会把自己搞疯。相比你做了哪些假设，更重要的是认识到自己做出了假设。不论是在实际操作还是面试中，就算假设条件不正确，只要可以识别出来，就能予以妥善处理。

此外，请记住，弹性和复杂度之间需要权衡取舍。若建立的系统支持一门课程可由多位教授来教，的确会增加数据库的弹性，但又徒增其复杂度。倘若要让数据库灵活应对各种可能的情况，最终数据库只会变得复杂不堪。

尽量让你的设计保持合理的弹性，并陈明任何其他的假设或限制条件。这不仅适用于数据库设计，对于面向对象设计和常规的编程同样适用。

9.16　线程与锁

16.1　线程和进程有何区别？（第 103 页）

解法

进程和线程彼此有关联，但两者有着根本上的不同。

进程可以看作是程序执行时的实例，是一个分配了系统资源（比如CPU时间和内存）的独立实体。每个进程都在各自独立的地址空间里执行，一个进程无法访问另一个进程的变量和数据结构。如果一个进程想要访问其他进程的资源，就必须使用进程间通信机制，包括管道、文件、套接字（socket）及其他形式。

线程存在于进程中，共享进程的资源（包括它的堆空间）。同一进程里的多个线程将共享同一个堆空间。这跟进程大不相同，一个进程不能直接访问另一个进程的内存。不过，每个线程仍然会有自己的寄存器和栈，而其他线程可以读写堆内存。

线程是进程的某条执行路径。当某个线程修改进程资源时，其他兄弟线程就会立即看到由此产生的变化。

16.2　如何测量上下文切换时间？（第 103 页）

解法

此题比较棘手，我们不妨先从一种可能的解法入手。

上下文切换（context switch）是两个进程之间切换（也即，将等待中的进程转为执行状态，而将正在执行的进程转为等待或终止状态）所耗费的时间。这样的动作会发生在多任务处理系统中，操作系统必须将等待中进程的状态信息载入内存，并保存执行中进程的状态信息。

为了解决此题，我们需要记录两个交换进程执行最后一条和第一条指令的时间戳，而上下文切换时间就是这两个进程的时间戳差值。

举个简单的例子：假设只有两个进程P_1和P_2。

P_1正在执行，P_2则在等待执行。在某一时间点，操作系统必须交换P_1和P_2，假设正好发生在P_1执行第N条指令之际。若$t_{x,k}$表示进程x执行第k条指令的时间戳，单位为微秒，则上下文切换需要$t_{2,1} - t_{1,n}$微秒。

此题棘手的地方在于：如何知道两个进程何时会进行交换呢？当然，我们无法记录进程每条指令的时间戳。

还有一个问题，进程交换是由操作系统的调度算法负责的，另外还可能有很多内核态线程也会进行上下文切换。其他进程也可能会竞争CPU，或者内核还要处理中断，用户控制不了这些不相干的上下文切换。举例来说，若内核在$t_{1,n}$时刻决定处理某个中断，那么，上下文切换时间就会比预估的更长。

为克服这些障碍，我们必须先构造一个环境：在P_1执行之后，任务调度器会立即选中并执行

P_2。具体做法是在P_1和P_2之间构造一条数据通道，如管道，让这两个进程玩一场数据令牌的桌球游戏。

换言之，我们让P_1作为初始发送方，P_2作为接收方。一开始，P_2阻塞（睡眠）等待获取数据令牌。P_1执行时会将令牌通过数据通道递送给P_2，并立即尝试读取响应令牌。然而，由于P_2还没有机会执行，因此P_1收不到这个响应令牌，继而被阻塞并释放CPU。

随之而来的就是上下文切换，任务调度器必须选择另一个进程执行。P_2正好处于随时可执行的状态，因此也就顺理成章地成为任务调度器可选择执行的理想候选者。当P_2执行时，P_1和P_2的角色互换了。现在，P_2成为发送方，而P_1成为被阻塞的接收方。当P_2将令牌返回给P_1时，游戏即告结束。

简而言之，这个游戏一个来回由以下步骤组成。

(1) P_2阻塞，等待P_1发送的数据。

(2) P_1标记开始时间。

(3) P_1向P_2发送令牌。

(4) P_1试着读取P_2发送的响应令牌，引发上下文切换。

(5) P_2被调度执行，接收P_1发送的令牌。

(6) P_2向P_1发送响应令牌。

(7) P_2试着读取P_1发送的响应令牌，引发上下文切换。

(8) P_1被调度执行，接收P_2发送的令牌。

(9) P_1标记结束时间。

这里的关键在于数据令牌的发送会引发上下文切换。令T_d和T_r分别为发送和接收数据令牌的时间，并令T_c为上下文切换耗费的时间。在第(2)步，P_1会记录令牌发送的时间戳，而在第(9)步则记录了令牌响应的时间戳。这两个事件之间用掉的时间T表示如下：

$$T = 2 * (T_d + T_c + T_r)$$

这个算式由以下事件组成：P_1发送一个令牌(3)，CPU上下文切换(4)，P_2接收这个令牌(5)。随后，P_2发送响应令牌(6)，CPU上下文切换(7)，最后P_1收到这个响应令牌(8)。

接着，由P_1很容易就能计算T，即事件3和事件8之间经过的时间。总之，若想求出T_c，我们必须先确定$T_d + T_r$的值。

该怎么做呢？我们可以测量P_1发送和接收令牌所耗费的时间多少。不过这不会引发上下文切换，因为发送这个令牌时P_1正在CPU中执行，而且接收时也不会处于阻塞状态。

将上述游戏重复玩多个来回，以剔除步骤(2)和(9)之间可能因意料之外的内核中断和其他内核线程对CPU的竞争而引入的时间变动。我们将选择测得的最短上下文切换时间作为最终答案。

话说回来，最后我们只能说，这只是近似值，而且取决于底层系统。比如，我们做了这样的假设：一旦数据令牌可用，P_2就会被选中并执行。而实际上，这要取决于任务调度器的具体实现，我们无法做出任何保证。

没关系，就算这样也不要紧。在面试中，能够意识到你的解法或许不够完美，这一点很重要。

16.3　在著名的哲学家就餐问题中，一群哲学家围坐在圆桌周围，每两位哲学家之间有一根筷子。每位哲学家需要两根筷子才能用餐，并且一定会先拿起左手边的筷子，然后才会去拿右手边的筷子。如果所有哲学家在同一时间拿起左手边的筷子，就有可能造成死锁。请使用线程和锁，编写代码模拟哲学家就餐问题，避免出现死锁。（第 103 页）

解法

首先，先不管死锁，让我们写些代码简单模拟哲学家就餐问题。具体实现时，从`Thread`派生`Philosopher`，`Chopstick`被拿起来时会调用`lock.lock()`，放下时则调用`lock.unlock()`。

```
public class Chopstick {
  private Lock lock;

  public Chopstick() {
    lock = new ReentrantLock();
  }

  public void pickUp() {
    void lock.lock();
  }

  public void putDown() {
    lock.unlock();
  }
}

public class Philosopher extends Thread {
  private int bites = 10;
  private Chopstick left;
  private Chopstick right;

  public Philosopher(Chopstick left, Chopstick right) {
    this.left = left;
    this.right = right;
  }

  public void eat() {
    pickUp();
    chew();
    putDown();
  }

  public void pickUp() {
    left.pickUp();
    right.pickUp();
  }

  public void chew() { }

  public void putDown() {
    left.putDown();
    right.putDown();
```

```
  }

  public void run() {
    for (int i = 0; i < bites; i++) {
      eat();
    }
  }
}
```

如果所有哲学家都拿起左手边的一根筷子，并都等着拿右手边的另一根筷子，运行上面的代码就可能造成死锁。

为了防止发生死锁，我们的实现可以采用如下策略：如有哲学家拿不到右手边的筷子，就让他放下已拿到的左手边的筷子。

```
public class Chopstick {
  /* 同前 */

  public boolean pickUp() {
    return lock.tryLock();
  }
}

public class Philosopher extends Thread {
  /* 同前 */

  public void eat() {
    if (pickUp()) {
      chew();
      putDown();
    }
  }

  public boolean pickUp() {
    /* 试着拿起筷子 */
    if (!left.pickUp()) {
      return false;
    }
    if (!right.pickUp()) {
      left.putDown();
      return false;
    }
    return true;
  }
}
```

在上面的代码里，要确保拿不到右手边的筷子时就要放下左手边的筷子；如果手上根本没有筷子，就不该调用putDown()。

16.4　设计一个类，只有在不可能发生死锁的情况下，才会提供锁。（第 103 页）

解法

防止死锁有几种常见的方法，其中常用的做法之一是，要求进程事先声明它需要哪些锁。然

后，就可以加以验证，若提供锁是否会造成死锁，会的话就不提供。

谨记这些限制条件，下面来探讨如何检测死锁。假设多个锁被请求的顺序如下：

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

这可能会造成死锁，因为，存在以下场景：

```
A锁住2，等待3
B锁住3，等待5
C锁住5，等待2
```

我们可以将上面的场景看作一个图，其中2连接到3、3连接到5，而5连接到2。死锁会由环表示。如果某个进程声明它会在锁住w后立即请求锁v，则图里就会存在一条边(w, v)。以先前的例子来说，在图里会存在下面这些边：(1, 2)、(2, 3)、(3, 4)、(1, 3)、(3, 5)、(7, 5)、(5, 9)、(9, 2)。至于这些边的“所有者”是谁并不重要。

这个类需要一个declare方法，线程和进程会以该方法声明它们请求资源的顺序。这个declare方法将迭代访问声明顺序，将邻近的每对元素(v, w)加到图里。然后，它会检查是否存在环。如果存在环，它就会原路返回，从图中移除这些边，然后退出。

现在只剩下一部分有待探讨：如何检测有无环？我们可以通过对每个连接起来的部分（也就是图中每个连接在一起的部分）执行深度优先搜索来检测有没有环。也有算法能选择图中所有连接的部分，但那样就会更复杂了。就此题而言，还没必要复杂到这个程度。

我们可以确定，如果出现了环，就表明是某一条新加入的边造成的。这样一来，只要深度优先搜索会探测所有这些边，就等同于做过完整的搜索。

这种特殊的环的检测算法，其伪码如下所示：

```
boolean checkForCycle(locks[] locks) {
   touchedNodes = hash table(lock -> boolean)
   initialize touchedNodes to false for each lock in locks
   for each (lock x in process.locks) {
     if (touchedNodes[x] == false) {
        if (hasCycle(x, touchedNodes)) {
           return true;
        }
     }
   }
   return false;
}

boolean hasCycle(node x, touchedNodes) {
   touchedNodes[r] = true;
   if (x.state == VISITING) {
     return true;
   } else if (x.state == FRESH) {
     ... (see full code below)
   }
}
```

注意，在上面的代码中，可能需要执行几次深度优先搜索，但touchedNodes只会初始化一次。我们会不断迭代，直至touchedNodes中所有值都变为false。

下面的代码提供了更多细节。为了简单起见，我们假设所有锁和进程（所有者）都是按顺序排列的。

```
public class LockFactory {
   private static LockFactory instance;

   private int numberOfLocks = 5; /* 缺省 */
   private LockNode[] locks;

   /* 从一个进程或所有者映射到该
    * 所有者宣称它会要求锁的顺序 */
   private Hashtable<Integer, LinkedList<LockNode>> lockOrder;

   private LockFactory(int count) { ... }
   public static LockFactory getInstance() { return instance; }

   public static synchronized LockFactory initialize(int count) {
     if (instance == null) instance = new LockFactory(count);
     return instance;
   }

   public boolean hasCycle(
        Hashtable<Integer, Boolean> touchedNodes,
        int[] resourcesInOrder) {
     /* 检查有无环 */
     for (int resource : resourcesInOrder) {
        if (touchedNodes.get(resource) == false) {
           LockNode n = locks[resource];
           if (n.hasCycle(touchedNodes)) {
              return true;
           }
        }
     }
     return false;
   }

   /* 为了避免死锁，强制每个进程都要事先宣告
    * 它们要求锁的顺序。验证这个顺序不会形成
    * 死锁（在有向图里出现
    * 环） */
   public boolean declare(int ownerId, int[] resourcesInOrder) {
     Hashtable<Integer, Boolean> touchedNodes =
        new Hashtable<Integer, Boolean>();

     /* 将结点加入图中 */
     int index = 1;
     touchedNodes.put(resourcesInOrder[0], false);
     for (index = 1; index < resourcesInOrder.length; index++) {
        LockNode prev = locks[resourcesInOrder[index - 1]];
        LockNode curr = locks[resourcesInOrder[index]];
```

```
        prev.joinTo(curr);
        touchedNodes.put(resourcesInOrder[index], false);
     }

     /* 如果出现了环，销毁这份资源列表，并
      * 返回false */
     if (hasCycle(touchedNodes, resourcesInOrder)) {
        for (int j = 1; j < resourcesInOrder.length; j++) {
           LockNode p = locks[resourcesInOrder[j - 1]];
           LockNode c = locks[resourcesInOrder[j]];
           p.remove(c);
        }
        return false;
     }

     /* 为检测到环，保存宣告的顺序，以便
      * 验证该进程确实按照它宣称的顺序要求
      * 锁 */
     LinkedList<LockNode> list = new LinkedList<LockNode>();
     for (int i = 0; i < resourcesInOrder.length; i++) {
        LockNode resource = locks[resourcesInOrder[i]];
        list.add(resource);
     }
     lockOrder.put(ownerId, list);

     return true;
   }

   /* 取得锁，首先验证该进程确实按照它宣告的顺序
    * 要求锁 */
   public Lock getLock(int ownerId, int resourceID) {
     LinkedList<LockNode> list = lockOrder.get(ownerId);
     if (list == null) return null;

     LockNode head = list.getFirst();
     if (head.getId() == resourceID) {
        list.removeFirst();
        return head.getLock();
     }
     return null;
   }
}

public class LockNode {
   public enum VisitState { FRESH, VISITING, VISITED };

   private ArrayList<LockNode> children;
   private int lockId;
   private Lock lock;
   private int maxLocks;

   public LockNode(int id, int max) { ... }

   /* 连接"this"结点与"node"结点，检查以确保这么做不会
```

```
   * 形成环 */
  public void joinTo(LockNode node) { children.add(node); }
  public void remove(LockNode node) { children.remove(node); }

  /* 以深度优先搜索检查是否存在环 */
  public boolean hasCycle(
       Hashtable<Integer, Boolean> touchedNodes) {
    VisitState[] visited = new VisitState[maxLocks];
    for (int i = 0; i < maxLocks; i++) {
       visited[i] = VisitState.FRESH;
    }
    return hasCycle(visited, touchedNodes);
  }

  private boolean hasCycle(VisitState[] visited,
       Hashtable<Integer, Boolean> touchedNodes) {
    if (touchedNodes.containsKey(lockId)) {
       touchedNodes.put(lockId, true);
    }

    if (visited[lockId] == VisitState.VISITING) {
       /* 还在访问时却回到了这个结点,
        * 表明有环 */
       return true;
    } else if (visited[lockId] == VisitState.FRESH) {
       visited[lockId] = VisitState.VISITING;
       for (LockNode n : children) {
          if (n.hasCycle(visited, touchedNodes)) {
             return true;
          }
       }
       visited[lockId] = VisitState.VISITED;
    }
    return false;
  }

  public Lock getLock() {
    if (lock == null) lock = new ReentrantLock();
    return lock;
  }

  public int getId() { return lockId; }
}
```

如同以往，当你看到这段既复杂又冗长的代码时，就会明白面试官一般不会要求你写出全部代码。更有可能的情况是，面试官会要求你勾勒出伪码，并实现其中一个方法。

16.5　给定以下代码：

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

同一个Foo实例会被传入 3 个不同的线程。threadA会调用first，threadB会调用second，threadC会调用third。设计一种机制，确保first会在second之前调用，second会在third之前调用。（第 103 页）

解法

一般的逻辑是检查在执行second()之前first()是否已完成，在调用third()之前second()是否已完成。我们必须小心处理线程安全，因此，简单的布尔标志达不到要求。

那么，以如下代码来使用锁，怎么样？

```
public class FooBad {
   public int pauseTime = 1000;
   public ReentrantLock lock1, lock2, lock3;

   public FooBad() {
     try {
        lock1 = new ReentrantLock();
        lock2 = new ReentrantLock();
        lock3 = new ReentrantLock();

        lock1.lock();
        lock2.lock();
        lock3.lock();
     } catch (...) { ... }
   }

   public void first() {
     try {
        ...
        lock1.unlock(); // 标记first()已完成
     } catch (...) { ... }
   }

   public void second() {
     try {
        lock1.lock(); // 等待，直到first()完成
        lock1.unlock();
        ...

        lock2.unlock(); // 标记second()已完成
     } catch (...) { ... }
   }

   public void third() {
     try {
        lock2.lock(); // 等待，直到second()完成
        lock2.unlock();
        ...
     } catch (...) { ... }
   }
}
```

这段代码实际上并不满足题目要求，关键在于锁的所有权这个概念。真正请求锁的是一个线程（在FooBad构造函数中），而释放锁的却是另一个线程。这么做是不允许的，这段代码会抛出异常。在Java中，锁的所有者和拿到锁的线程必须是同一个。

换种做法，我们可以用信号量重现这一行为，整个逻辑完全相同。

```
public class Foo {
   public Semaphore sem1, sem2, sem3;

   public Foo() {
     try {
        sem1 = new Semaphore(1);
        sem2 = new Semaphore(1);
        sem3 = new Semaphore(1);

        sem1.acquire();
        sem2.acquire();
        sem3.acquire();
     } catch (...) { ... }
   }

   public void first() {
     try {
        ...
        sem1.release();
     } catch (...) { ... }
   }

   public void second() {
     try {
        sem1.acquire();
        sem1.release();
        ...
        sem2.release();
     } catch (...) { ... }
   }

   public void third() {
     try {
        sem2.acquire();
        sem2.release();
        ...
     } catch (...) { ... }
   }
}
```

16.6　给定一个类，内含同步方法A和普通方法B。在同一个程序实例中，有两个线程，能否同时执行A？两者能否同时执行A和B？（第 104 页）

解法

在方法前加上关键字synchronized，即可保证两个线程无法同时执行某个对象的同步方法。

因此，第一个子问题的答案要视具体情况而定。如果两个线程拥有该对象的同一实例，那么，答案就是否定的，它们不能同时执行方法A。不过，要是这两个线程拥有该对象的不同实例，就能同时执行方法A。

在概念上，你可以从“锁”的角度来考虑答案。同步方法会对所属对象特定实例的所有同步方法上锁，从而阻止任何其他线程执行那个实例的同步方法。

第二个子问题问的是，thread2在执行非同步方法B时，thread1能否执行同步方法A。既然B不是同步方法，在thread2执行方法B时，也就无从阻止thread1执行方法。不管thread1和thread2是否拥有该对象的同一实例，这一点都成立。

说到底，此题强调的关键概念是，那个对象的每个实例只能执行一个同步方法。其他线程可以执行该实例的非同步方法，或者，它们可以执行该对象不同实例的任意方法。

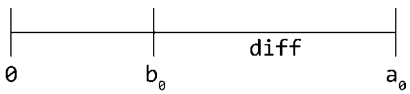
9.17　中等难题

17.1　编写一个函数，不用临时变量，直接交换两个数。（第 104 页）

解法

这是个经典面试题，也相当直接。我们将用a_0表示a的初始值，b_0表示b的初始值，用diff表示a_0 - b_0的值。

让我们将a > b的情形绘制在数轴上。



首先，将a设为diff，即上面数轴的右边那一段。然后，b加上diff（并将结果保存在b中），就可得到a_0。至此，我们得到b = a_0和a = diff。最后，只需将b设为a_0 - diff，也就是b - a。

下面是具体的实现代码。

```
public static void swap(int a, int b) {
   // 以a = 9、b = 4为例
   a = a - b; // a = 9 - 4 = 5
   b = a + b; // b = 5 + 4 = 9
   a = b - a; // a = 9 - 5

   System.out.println(“a: ” + a);
   System.out.println(“b: ” + b);
}
```

我们还可以用位操作实现类似的解法，这种解法的优点在于它适用的数据类型更多，不仅限于整数。

```
public static void swap_opt(int a, int b) {
   // 以a = 101（二进制）和b = 110为例
   a = a^b; // a = 101^110 = 011
```

```
    b = a^b; // b = 011^110 = 101
    a = a^b; // a = 011^101 = 110

    System.out.println("a: " + a);
    System.out.println("b: " + b);
}
```

这段代码使用了异或操作,要了解个中细节,最简单的方法就是看看两个比特位p和q的情况,一探究竟。这里会用p_0和q_0表示初始值。

如能正确交换两个比特位，整个操作就能正确无误地进行。下面将逐行解析交换过程。

```
1  p = p0^q0  /* 若p0 = q0则为0，若p0 != q0则为1 */
2  q = p^q0   /* 等于p0的值 */
3  p = p^q    /* 等于q0的值 */
```

第1行执行操作$p = p_0 \wedge q_0$，若$p_0 = q_0$则结果为0；若$p_0 != q_0$则为1。

第2行执行$q = p \wedge q_0$，可以就p为0和1两种可能的值进行检查。最终目的是要交换p和q的初始值，我们希望这个操作返回p_0的值。

- 若p = 0：则$p_0 = q_0$，因此，我们需要该操作返回p_0或q_0。任意值与0异或都会返回初始值，由此可知该操作会正确返回q_0（或p_0）。
- 若p = 1：则$p_0 != q_0$。我们希望该操作，当q_0为0时返回1，p_0为1时返回0。这正是将任意值与1执行异或操作的结果。

第3行执行p = p^q，再次检查p为0和1两种值的情况，目的是返回q_0。注意，q现在等于p_0，因此其实是在执行$p \wedge p_0$。

- 若p = 0：由于$p_0 = q_0$，我们希望该操作返回p_0或q_0，不论哪一个都可以。执行$0 \wedge p_0$会返回p_0，等于q_0。
- 若p = 1：该操作其实是在执行$1 \wedge p_0$。这会翻转p_0的值，而这正是我们想要的，因为$p_0 != q_0$。

至此，我们已将p设为q_0，q设为p_0。综上，上述操作会正确交换两个比特位，因此，就能正确交换整个整数。

17.2 设计一个算法，判断玩家是否赢了井字游戏。（第 104 页）

解法

乍一看，可能会觉得此题真的很简单，不就是直接检查井字棋盘，这会有多难呢？细一想，此题还是有点复杂的，而且没有唯一的“完美”答案。根据你的喜好不同，会有不一样的最佳解法。

解决此题，有几个重要的设计决策需要考虑。

(1) hasWon只会调用一次还是很多次（比如，放在网站上的井字游戏）？如果答案是后者，我们可能会增加一些预处理，以优化hasWon的运行时间。

(2) 井字游戏通常是3×3棋盘。我们只是针对3×3大小的棋盘进行设计，还是要实现一个$N \times N$的解法？

(3) 对于程序大小、执行速度和代码清晰度，一般如何区分它们的优先级呢？记住，最高效的代码不一定是最好的。代码是否容易理解、维护也很重要。

解法 1：如果 hasWon 会被调用很多次

总共只有3^9，大约20 000种井字游戏棋盘（假设为3×3的棋盘）。因此，用一个`int`就能表示，其中每个数位代表棋盘中的一格（0为空、1为红、2为蓝）。我们会事先设定好一个散列表或数组，将所有可能的棋盘作为键，值则代表谁赢了。这么一来，`hasWon`函数就很简单了：

```
public int hasWon(int board) {
   return winnerHashtable[board];
}
```

要将一个棋盘（以字符数组表示）转成一个`int`，可以运用“3进位”表示法，每个棋盘可表示为$3^0v_0 + 3^1v_1 + 3^2v_2 + \dots + 3^8v_8$，若格子为空则$v_i$为0，格子为蓝色则$v_i$为1，格子为红色则$v_i$为2。

```
public static int convertBoardToInt(char[][] board) {
   int factor = 1;
   int sum = 0;
   for (int i = 0; i < board.length; i++) {
     for (int j = 0; j < board[i].length; j++) {
        int v = 0;
        if (board[i][j] == 'x') {
           v = 1;
        } else if (board[i][j] == 'o') {
           v = 2;
        }
        sum += v * factor;
        factor *= 3;
     }
   }
   return sum;
}
```

至此，要判断谁是赢家，只需查询散列表即可。

当然，如果每次判断谁赢了都要将棋盘转成这种格式，那么跟其他解法相比，其实并没有节省多少时间。但是，如果一开始就以这种格式存储棋盘，那么，查询操作将会非常有效率。

解法 2：专为 3×3 棋盘设计

如果只想为3×3棋盘设计一种解法，代码就会比较简短且简单。复杂的地方只剩下如何写得清晰而有条理，并且不要写出太多重复代码。

```
Piece hasWon1(Piece[][] board) {
   for (int i = 0; i < board.length; i++) {
     /* 检查行 */
     if (board[i][0] != Piece.Empty &&
        board[i][0] == board[i][1] &&
        board[i][0] == board[i][2]) {
         return board[i][0];
     }

     /* 检查列 */
     if (board[0][i] != Piece.Empty &&
```

```
        board[0][i] == board[1][i] &&
        board[0][i] == board[2][i]) {
       return board[0][i];
    }
  }

  /* 检查对角线 */
  if (board[0][0] != Piece.Empty &&
     board[0][0] == board[1][1] &&
     board[0][0] == board[2][2]) {
    return board[0][0];
  }

  /* 检查逆对角线 */
  if (board[2][0] != Piece.Empty &&
     board[2][0] == board[1][1] &&
     board[2][0] == board[0][2]) {
    return board[2][0];
  }
  return Piece.Empty;
}
```

解法 3：面向 $N \times N$ 棋盘进行设计

有了3×3棋盘的实现代码，自然就会想到要扩展到$N \times N$棋盘。本书可下载的源码提供了另外几种解法，下面是其中一种。

```
Piece hasWon3(Piece[][] board) {
  int N = board.length;
  int row = 0;
  int col = 0;

  /* 检查行 */
  for (row = 0; row < N; row++) {
    if (board[row][0] != Piece.Empty) {
       for (col = 1; col < N; col++) {
          if (board[row][col] != board[row][col-1]) break;
       }
       if (col == N) return board[row][0];
    }
  }

  /* 检查列 */
  for (col = 0; col < N; col++) {
    if (board[0][col] != Piece.Empty) {
       for (row = 1; row < N; row++) {
          if (board[row][col] != board[row-1][col]) break;
       }
       if (row == N) return board[0][col];
    }
  }

  /* 检查对角线（左上到右下） */
  if (board[0][0] != Piece.Empty) {
```

```
    for (row = 1; row < N; row++) {
      if (board[row][row] != board[row-1][row-1]) break;
    }
    if (row == N) return board[0][0];
  }

  /* 检查对角线（左下到右上） */
  if (board[N-1][0] != Piece.Empty) {
    for (row = 1; row < N; row++) {
      if (board[N-row-1][row] != board[N-row][row-1]) break;
    }
    if (row == N) return board[N-1][0];
  }

  return Piece.Empty;
}
```

不论你的解法为何，此题的算法并不是太难。重点在于理解如何写出清晰、可维护的代码，而这也正是面试官想要评估的地方。

17.3　设计一个算法，算出*n*阶乘有多少个尾随零。（第 104 页）

解法

简单的做法是先算出阶乘，然后不断地除以10，数一数有几个尾随零（trailing zero）。但这种做法的问题是，使用int很快就会越界。为了避开这个限制，我们可以从数学上来分析这个问题。

下面以阶乘19!为例进行说明：

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

10倍数就会形成尾随零，而10倍数又可分解为一组组5倍数和2倍数。

例如，在19!中，下列几项会形成尾随零：

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

因此，为了算出尾随零的数量，我们只需计算有几对5和2倍数。不过，2倍数始终要比5倍数多，最后只要数出5倍数就可以了。

这里有个陷阱，就是15只能算一个5倍数（因此会形成一个尾随零），而25算两个（25 = 5 * 5）。

编写代码时，相关代码有两种写法。

第一种写法是迭代访问所有2到*n*的数字，计算每个数字中有几个5。

```
/* 若数字为5的倍数，返回5的几次方
 *    5 -> 1,
 *    25-> 2等
*/
public int factorsOf5(int i) {
  int count = 0;
  while (i % 5 == 0) {
    count++;
    i /= 5;
```

```
  }
  return count;
}

public int countFactZeros(int num) {
  int count = 0;
  for (int i = 2; i <= num; i++) {
    count += factorsOf5(i);
  }
  return count;
}
```

这么写还不赖，不过，我们还可以做得更有效率一点：直接数一数5的因数。采用这种做法，我们会先数一数1到n之间，有几个5的倍数（数量为$n/5$），然后数一数25的倍数有几个（$n/25$），接着是125，依此类推。

要算出n中有几个m的倍数，直接将n除以m即可。

```
public int countFactZeros(int num) {
  int count = 0;
  if (num < 0) {
    return -1;
  }
  for (int i = 5; num / i > 0; i *= 5) {
    count += num / i;
  }
  return count;
}
```

此题有点像脑筋急转弯，不过，还是可以通过逻辑思考来解决（如上所示）。只要思考一下到底有哪些条件会形成尾随零，就能得到解法。你必须从一开始就透彻地理解相关规则，才能正确地实现出来。

17.4　编写一个方法，找出两个数字中最大的那一个。不得使用if-else或其他比较运算符。（第 104 页）

解法

max函数的常见实现方式是检查$a - b$的正负号。但这里不能使用比较运算符检查正负情况，不过我们可以使用乘法。

假定k代表$a - b$的正负号，如果 $a - b >= 0$，则k为1，否则k为0。令q为k的反数。

那么，我们可以实现如下代码：

```
/* 1变0，0变1 */
public static int flip(int bit) {
  return 1^bit;
}

/* a为正则返回1，a为负则返回0 */
public static int sign(int a) {
  return flip((a >> 31) & 0x1);
```

```
}

public static int getMaxNaive(int a, int b) {
   int k = sign(a - b);
   int q = flip(k);
   return a * k + b * q;
}
```

这段代码看似可行，实则不济。要是$a-b$溢出，这段代码就行不通。例如，假设a为`INT_MAX - 2`，b为`-15`。此时，$a-b$将大于`INT_MAX`并且会溢出，最终变为负值。

运用同样的方法，我们可以实现此题的解法，目标是当$a>b$时维持k为1的条件。为此，我们需要使用更为复杂的逻辑。

$a-b$什么时候会溢出呢？它只会在a为正、b为负时溢出，或者，反过来也有可能。专门检测溢出条件可能比较困难，不过，我们可以检测a和b何时会有不同的正负号。注意，如果a和b的正负号不同，就让k等于`sign(a)`。

具体逻辑如下：

```
if a和b的正负号不同:
   // 若a > 0，则b < 0且k = 1
   // 若a < 0，则b > 0且k = 0
   // 因此，不管哪种情况，k = sign(a)
   let k = sign(a)
else
   let k = sign(a - b) // 这里不再有溢出
```

上述逻辑的实现代码如下，其中使用了乘法而不是if语句。

```
public static int getMax(int a, int b) {
   int c = a - b;

   int sa = sign(a); // if a >= 0, then 1 else 0
   int sb = sign(b); // if b >= 0, then 1 else 0
   int sc = sign(c); // 取决于a - b有没有溢出

   /* 目标：定义k的值，若a > b则为1，a < b则为0
    * （若a = b，k为何值无关紧要） */

   // 若a和b正负号不同，则k = sign(a)
   int use_sign_of_a = sa ^ sb;

   // 若a和b正负号相同，则k = sign(a - b)
   int use_sign_of_c = flip(sa ^ sb);

   int k = use_sign_of_a * sa + use_sign_of_c * sc;
   int q = flip(k); // k的反数

   return a * k + b * q;
}
```

注意，为清晰起见，我们将代码拆分成多个方法和变量。很显然，这不是最紧凑或最有效率的写法，但这么写代码要清晰许多。

17.5　珠玑妙算游戏（The Game of Master Mind）的玩法如下。

计算机有四个槽，每个槽放一个球，颜色可能是红色（R）、黄色（Y）、绿色（G）或蓝色（B）。例如，计算机可能有RGGB四种（槽 1 为红色，槽 2、3 为绿色，槽 4 为蓝色）。

作为用户，你试图猜出颜色组合。打个比方，你可能会猜YRGB。

要是猜对某个槽的颜色，则算一次“猜中”；要是只猜对颜色但槽位猜错了，则算一次“伪猜中”。注意，“猜中”不能算入“伪猜中”。

举个例子，实际颜色组合为RGBY，而你猜的是GGRR，则算一次猜中，一次伪猜中。

给定一个猜测和一种颜色组合，编写一个方法，返回猜中和伪猜中的次数。（第 104 页）

解法

此题简单明了，但令人惊讶的是，写代码时很容易犯小错误。代码写好后，你应该对照各种测试用例，进行全面彻底的检查。

编写代码时，我们首先会构造一个频率数组，存放每个字符在solution中出现的次数，但不包括某个槽被“猜中”的次数。然后，迭代guess算出伪猜中的次数。

下面是这个算法的实现代码。

```
public class Result {
   public int hits = 0;
   public int pseudoHits = 0;

   public String toString() {
     return “(“ + hits + “, “ + pseudoHits + “)”;
   }
}

public int code(char c) {
   switch (c) {
   case ‘B’:
     return 0;
   case ‘G’:
     return 1;
   case ‘R’:
     return 2;
   case ‘Y’:
     return 3;
   default:
     return -1;
   }
}

public static int MAX_COLORS = 4;

public Result estimate(String guess, String solution) {
   if (guess.length() != solution.length()) return null;

   Result res = new Result();
   int[] frequencies = new int[MAX_COLORS];

```

```
  /* 计算猜中次数，构造频率表 */
  for (int i = 0; i < guess.length(); i++) {
    if (guess.charAt(i) == solution.charAt(i)) {
       res.hits++;
    } else {
       /* 只有不是猜中的情况下，才增加频率表
        * （将用于伪猜中）。若是猜中，那么，
        * 该槽位已被“使用” */
       int code = code(solution.charAt(i));
       frequencies[code]++;
    }
  }

  /* 计算伪猜中 */
  for (int i = 0; i < guess.length(); i++) {
    int code = code(guess.charAt(i));
    if (code >= 0 && frequencies[code] > 0 &&
          guess.charAt(i) != solution.charAt(i)) {
       res.pseudoHits++;
       frequencies[code]--;
    }
  }
  return res;
}
```

注意，问题所需的算法越简单，写出清晰、正确的代码就越显重要。在上面的例子中，我们提取代码专门写了个`code(char c)`方法，并创建了一个`Result`类来保存结果，而非只是打印显示。

17.6 给定一个整数数组，编写一个函数，找出索引*m*和*n*，只要将*m*和*n*之间的元素排好序，整个数组就是有序的。注意：*n* – *m*越小越好，也就是说，找出符合条件的最短序列。**（第 104 页）**

解法

开始解题之前，让我们先确认一下答案会是什么样的。如果要找的是两个索引，这表明数组中间有一段有待排序，其中数组开头和末尾部分是排好序的。

现在，我们借用下面的例子来解决此题：

```
1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19
```

首先映入脑海的想法可能是，直接找出位于开头的最长递增子序列，以及位于末尾的最长递增子序列。

```
左边：1, 2, 4, 7, 10, 11
中间：7, 12
右边：6, 7, 16, 18, 19
```

很容易就能找出这些子序列，只需从数组最左边和最右边开始，向中间查找递增子序列。一旦发现有元素大小顺序不对，那就是找到了递增/递减子序列的两头。

但是，为了解决这个问题，还需要对数组中间部分进行排序，只要将中间部分排好序，数组所有元素便是有序的。具体来说，就是以下判断条件必须为真：

```
/* 左边（left）所有元素都要小于中间（middle）的所有元素 */
min(middle) > end(left)

/* 中间（middle）所有元素都要小于右边（right）的所有元素 */
max(middle) < start(right)
```

或者，换句话说，对于所有元素：

```
left < middle < right
```

实际上，上例的这个条件绝不可能成立。根据定义，中间部分的元素是无序的。而在上面的例子中，`left.end > middle.start`且`middle.end > right.start`一定成立。这样一来，只排序中间部分并不能让整个数组有序。

不过，我们还可以缩减左边和右边的子序列，直到先前的条件成立为止。

令`min`等于`min(middle)`，`max`等于`max(middle)`。

对左边部分，我们先从这个子序列的末尾开始（值为11，索引为5），并向左移动，直至找到元素索引*i*使得`array[i] < min`；找到后只需排序中间部分，就能让数组的那部分有序。

然后，对右边部分进行类似操作，此时`max`等于12。我们先从右边子序列的起始元素（值为6）开始，并向右移动，将中间部分的最大值12依次与6、7、16比较。找到16时，就能确定在16的右边已经没有元素比12小了（因为右边是递增子序列）。至此，对数组中间部分进行排序，以使整个数组都是有序的。

下面是这个算法的实现代码。

```java
int findEndOfLeftSubsequence(int[] array) {
   for (int i = 1; i < array.length; i++) {
     if (array[i] < array[i - 1]) return i - 1;
   }
   return array.length - 1;
}

int findStartOfRightSubsequence(int[] array) {
   for (int i = array.length - 2; i >= 0; i--) {
     if (array[i] > array[i + 1]) return i + 1;
   }
   return 0;
}

int shrinkLeft(int[] array, int min_index, int start) {
   int comp = array[min_index];
   for (int i = start - 1; i >= 0; i--) {
     if (array[i] <= comp) return i + 1;
   }
   return 0;
}

int shrinkRight(int[] array, int max_index, int start) {
```

```
  int comp = array[max_index];
  for (int i = start; i < array.length; i++) {
    if (array[i] >= comp) return i - 1;
  }
  return array.length - 1;
}

void findUnsortedSequence(int[] array) {
  /* 找出左子序列 */
  int end_left = findEndOfLeftSubsequence(array);

  /* 找出右子序列 */
  int start_right = findStartOfRightSubsequence(array);

  /* 找出中间部分的最小值和最大值 */
  int min_index = end_left + 1;
  if (min_index >= array.length) return; // 已排序

  int max_index = start_right - 1;
  for (int i = end_left; i <= start_right; i++) {
    if (array[i] < array[min_index]) min_index = i;
    if (array[i] > array[max_index]) max_index = i;
  }

  /* 向左移动，直到小于array[min_index] */
  int left_index = shrinkLeft(array, min_index, end_left);

  /* 向右移动，直到大于array[max_index] */
  int right_index = shrinkRight(array, max_index, start_right);

  System.out.println(left_index + “ “ + right_index);
}
```

注意，在上面的解法中，我们还创建了不少方法。虽然也可以把所有代码一股脑儿塞进一个方法，但这样一来，代码理解、维护和测试起来就要难得多。在面试中写代码时，你应该优先考虑这几点。

17.7　给定一个整数，打印该整数的英文描述（例如“One Thousand, Two Hundred-Thirty Four”）。（第 104 页）

解法

此题并不太难，反倒有点乏味。关键在于解题的过程和组织，并确定你有完善的测试用例。

举个例子，在转换19 323 984时，我们可以考虑分段处理，每三位转换一次，并在适当的地方插入“thousand”（千）和“million”（百万）。也即，

```
convert(19 323 984) =  convert(19) + “ million ” +
                       convert(323) + “ thousand ” +
                       convert(984)
```

下面是该算法的实现代码。

```
public String[] digits = {"One", "Two", "Three", "Four", "Five",
   "Six", "Seven", "Eight", "Nine"};
public String[] teens = {"Eleven", "Twelve", "Thirteen",
   "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen",
   "Nineteen"};
public static String[] tens = {"Ten", "Twenty", "Thirty", "Forty",
   "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
public static String[] bigs = {"", "Thousand", "Million"};

public static String numToString(int number) {
   if (number == 0) {
     return "Zero";
   } else if (number < 0) {
     return "Negative " + numToString(-1 * number);
   }

   int count = 0;
   String str = "";

   while (number > 0) {
     if (number % 1000 != 0) {
        str = numToString100(number % 1000) + bigs[count] +
              " " + str;
     }
     number /= 1000;
     count++;
   }

   return str;
}

public static String numToString100(int number) {
   String str = "";

   /* 转换百位数的地方 */
   if (number >= 100) {
     str += digits[number / 100 - 1] + " Hundred ";
     number %= 100;
   }

   /* 转换十位数的地方 */
   if (number >= 11 && number <= 19) {
     return str + teens[number - 11] + " ";
   } else if (number == 10 || number >= 20) {
     str += tens[number / 10 - 1] + " ";
     number %= 10;
   }

   /* 转换个位数的地方 */
   if (number >= 1 && number <= 9) {
     str += digits[number - 1] + " ";
   }

   return str;
}
```

9

处理这类问题的关键在于，因为有很多特殊情况，所以要确保考虑到所有特殊情况。

17.8　给定一个整数数组（有正数有负数），找出总和最大的连续数列，并返回总和。（第 104 页）

解法

此题难度不小，但又极为常见。接下来，我们会通过下面的例子来解题：

```
2    3   -8   -1    2    4   -2    3
```

如果把上面的数组看作是正数数列和负数数量交替出现，我们会发现，答案绝不会只包含某负数子数列或正数子数列的一部分。何以见得？只包含某负数子数列的一部分，将使得总和过小，我们应该排除整个负数数列才对。同样地，只包含正数子数列的一部分也会显得很怪，因为若包含整个子数列，总和就能变得更大。

为了构思出算法，我们可以把数组看作一个正负数交错出现的数列。每个数字代表正数子数列的总和，或负数子数列的总和。对于上面的数组，简化后如下：

```
5   -9    6   -2    3
```

我们无法从中立即窥得很棒的算法，不过，它确实可以帮助我们更好地理解手头正在处理的问题。

考虑上面的数组。把{5, -9}视作子数列说得通吗？不，这两个数字的总和为-4，所以最好两个数字都不要，或者考虑只包含子数列{5}，只有一个元素。

什么情况下需要在子数列中包含负数呢？只有当它能将两个正子数列拼接在一起，并且两者加起来大于这个负数的时候。

我们可以一步一步地找出答案，先从数组的第一个元素开始。

首先看到5，这是到目前为止最大的总和。我们将maxsum设为5，并将sum设为5。接着，考虑-9，将它与sum相加会得到负值。将子数列从5延伸到-9并没有意义（只会将子数列缩减为-4），因此我们会重置sum的值。

现在看到6，这个子数列比5大，因此更新maxsum和sum。

接着来看-2，与6相加，sum设为4。由于总和仍会变大（与其他部分连接时，会有更长的数列），我们有可能想把{6, -2}纳入最长子数列，因此更新sum，但不更新maxsum。

最后看到3，3加上sum（4）结果为7，更新maxsum，最后得到最长子数列为{6, -2, 3}。

推而广之，对于完全展开的数组而言，处理逻辑是一样的。下面是该算法的实现代码。

```java
public static int getMaxSum(int[] a) {
   int maxsum = 0;
   int sum = 0;
   for (int i = 0; i < a.length; i++) {
     sum += a[i];
     if (maxsum < sum) {
        maxsum = sum;
```

```
      } else if (sum < 0) {
         sum = 0;
      }
    }
    return maxsum;
}
```

如果整个数组都是负数，怎么样才是正确的行为？看看这个简单的数组：`{-3, -10, -5}`，以下答案每个都说得通：

(1) -3（假设子数列不能为空）;

(2) 0（子数列长度为零）;

(3) MINIMUM_INT（视为错误情况）。

我们会选择第二个（`maxsum = 0`），但其实并没有所谓的“正确”答案。这一点可以跟面试官好好讨论一番，这样也能展示出你是个注重细节的人。

17.9　设计一个方法，找出任意指定单词在一本书中的出现频率。(第 104 页)

解法

面对面试官，该问的第一个问题是，这个操作是执行一次还是不断被执行。也就是说，你是只要找出“dog”的频率，还是想找出“dog”，接着是“cat”、“mouse”，等等？

1. 解法：单次查询

在这种情况下，我们会直接逐字逐句地扫描整本书，数一数某个单词出现的次数，用时$O(n)$。可以确定这是最短用时，因为不管怎么样，我们必须查看过书中的每个单词。

2. 解法：重复查询

如果是要重复执行查询操作，那么，或许值得我们多花些时间，多分配内存，对全书进行预处理。我们可以构造一个散列表，将单词映射到该单词的出现频率，这么一来，任意单词的频率都能在$O(1)$时间内找到。具体实现代码如下。

```
Hashtable<String, Integer> setupDictionary(String[] book) {
    Hashtable<String, Integer> table =
      new Hashtable<String, Integer>();
    for (String word : book) {
      word = word.toLowerCase();
      if (word.trim() != “”) {
         if (!table.containsKey(word)) {
            table.put(word, 0);
         }
         table.put(word, table.get(word) + 1);
      }
    }
    return table;
}

int getFrequency(Hashtable<String, Integer> table, String word) {
```

```
  if (table == null || word == null) return -1;
  word = word.toLowerCase();
  if (table.containsKey(word)) {
    return table.get(word);
  }
  return 0;
}
```

注意，相对而言，这类问题还是比较容易的。因此，面试官会更看重你的心思有多缜密，有没有检查错误条件？

17.10 XML非常冗长，你找到一种编码方式，可将每个标签对应为预先定义好的整数值，该编码方式的语法如下：

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> 对应到某个预先定义好的整数值
Value     --> 字符串值 END
```

例如，下列XML会被转换压缩成下面的字符串（假定对应关系为`family -> 1`、`person -> 2`、`firstName -> 3`、`lastName -> 4`、`state -> 5`）。

```
<family lastName="McDowell" state="CA">
   <person firstName="Gayle">Some Message</person>
</family>
```

变为：

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0.
```

编写代码，打印XML元素编码后的版本（传入`Element`和`Attribute`对象）。（第 105 页）

解法

由题目可知，元素会以`Element`和`Attribute`作为参数传入，因此具体代码相当简单，可以运用类似树状结构的做法实现。

我们会不断对XML结构的各个部分调用`encode()`，根据XML元素的类型，处理方式稍有不同。

```
public static void encode(Element root, StringBuffer sb) {
  encode(root.getNameCode(), sb);
  for (Attribute a : root.attributes) {
    encode(a, sb);
  }
  encode("0", sb);
  if (root.value != null && root.value != "") {
    encode(root.value, sb);
  } else {
    for (Element e : root.children) {
       encode(e, sb);
    }
  }
  encode("0", sb);
}
```

```
16
17 public static void encode(String v, StringBuffer sb) {
18    sb.append(v);
19    sb.append(" ");
20 }
21
22 public static void encode(Attribute attr, StringBuffer sb) {
23    encode(attr.getTagCode(), sb);
24    encode(attr.value, sb);
25 }
26
27 public static String encodeToString(Element root) {
28    StringBuffer sb = new StringBuffer();
29    encode(root, sb);
30    return sb.toString();
31 }
```

请留意第17行代码，有个负责处理字符串的简单方法encode。这个方法似乎有点画蛇添足，它无非就是插入字符串并附加一个空格。不过，使用这个方法有个好处，可以确保每个元素之间都有空格。否则，很可能就会忘记附加空白符从而打乱编码。

17.11 给定rand5()，实现一个方法rand7()。也即，给定一个产生 0 到 4（含）随机数的方法，编写一个产生 0 到 6（含）随机数的方法。（第 105 页）

解法

这个函数要正确实现，则返回0到6之间的值，每个值的概率必须为1/7。

1. 第一次尝试（调用次数固定）

第一次尝试时，我们可能会想产生出0到9之间的值，然后再除以7取余数。代码大致如下：

```
int rand7() {
   int v = rand5() + rand5();
   return v % 7;
}
```

可惜的是，上面的代码无法以相同的概率产生所有值。分析一下每次调用rand5()返回的结果与rand7()函数返回值的对应关系，就能确认这一点。

因为每一行会调用两次rand5()，每次调用返回不同值的概率为1/5，所以，每一行出现的概率为1/25。数一数每个数字出现的次数，就会发现这个rand7()函数以5/25的概率返回4，而返回0的概率为3/25。也就是说，这个函数与题目要求不符，返回各种结果的概率并非1/7。

现在设想一下，若我们要修改上面的函数加上一条if语句，并修改常数乘数或再插入一个rand5()调用，同样会产生一张类似的表格，而每一行组合出现的概率将是$1/5^k$，其中k为那一行调用rand5()的次数。不同行调用rand5()的次数可能不同。

最终，rand7()函数返回结果的概率，比如6，为所有结果为6的行的概率总和，也就是：

$$P(\text{rand7}() = 6) = 1/5^i + 1/5^j + \cdots + 1/5^m$$

为了保证函数正确实现，这个概率必须等于1/7。

1st Call	2nd Call	Result
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
1	0	1
1	1	2
1	2	3
1	3	4
1	4	5
2	0	2
2	1	3
2	2	4
2	3	5
2	4	6
3	0	3
3	1	4
3	2	5
3	3	6
3	4	0
4	0	4
4	1	5
4	2	6
4	3	0
4	4	1

但这又不可能，因为5和7互质，5倒数的指数级数不可能得到1/7。

难道此题无解吗？并非如此。严格地说，这意味着，rand5()调用组合的结果若能得到rand7()的某个特定值，只要能列出来，该函数就不会返回均匀分布的结果。

我们还是有办法解出此题的，只不过必须使用while循环，另外请注意，我们无法确定返回一个结果要经过几次循环。

2. 第二次尝试（调用次数不定）

只要能使用while循环，工作就会变得简单许多。我们只需产生出一个范围的数值，且每个数值出现的概率相同（且这个范围至少要有7个元素）。如果能做到这一点，就可以舍弃后面大于7的倍数的部分，然后将余下元素除以7取余数。由此将得到范围0到6的值，且每个值出现的概率相等。

下面的代码会通过5 * rand5() + rand5()产生范围0到24。然后，舍弃21和24之间的数值，否则rand7()返回0到3的值就会偏多，最后除以7取余数，得到范围0到6的数值，每个值出现的概率相同。

注意，这种做法需要舍弃一些值，因此不确定返回一个值要调用几次rand5()，这就是所谓的调用次数不定。

```
public static int rand7() {
   while (true) {
     int num = 5 * rand5() + rand5();
     if (num < 21) {
        return num % 7;
     }
   }
}
```

注意，执行`5 * rand5() + rand5()`正好只提供了一种方式来取得范围0到24之间的每个数值，这就确保了每个值出现的概率相同。

可以换个做法执行`2 * rand5() + rand5()`吗？不行，因为这些值不是均匀分布的。例如，取得6有两种方式（`6=2*1+4`和`6 = 2*2+2`），而取得0（`0=2*0+0`）则只有一种方式，在范围里的值出现概率不等。

还有一种做法就是使用`2 * rand5()`，这样也能得到均匀分布的值，但要复杂得多。代码如下：

```
public int rand7() {
   while (true) {
     int r1 = 2 * rand5(); /* 0和9之间的偶数 */
     int r2 = rand5(); /* 之后会用来产生0或1 */
     if (r2 != 4) { /* r2有多余的偶数，舍弃之 */
        int rand1 = r2 % 2; /* 产生0或1 */
        int num = r1 + rand1; /* 将会在范围0到9之间 */
        if (num < 7) {
           return num;
        }
     }
   }
}
```

事实上，我们可以使用的范围是无限的。关键在于确保该范围足够大，且范围内所有值出现的概率相同。

17.12 设计一个算法，找出数组中两数之和为指定值的所有整数对。（第 105 页）

解法

此题有两种解法，至于哪一种“比较好”，取决于你在时间效率、空间效率和代码复杂度之间如何取舍。

1. 简单解法

这个解法简单且高效（时间上），使用一个整数到整数的散列映射。这个算法会迭代整个数组，对于元素x，在散列表中查找$\text{sum} - x$，若存在就打印$(x, \text{sum} - x)$。将x加入散列表，然后继续处理下一个元素。

2. 另一种解法

首先，让我们从定义入手。试着要找一对总和为z的数，则x的补数为$z - x$（也即，与x相加得z的数）。举个例子，若要找一对总和为12的数，那么，−5的补数为17。

现在，假设有这个已排好序的数组：`{-2 -1 0 3 5 6 7 9 13 14}`。令`first`指向数组开头，`last`指向数组结尾。要找出`first`的补数，就将`last`往回移动，直至找到补数。如果`first + last < sum`，则数组中不存在`first`的补数，因此可以向前移动`first`，等到`first`比`last`大时停止操作。

为什么这么做就能找出`first`的所有补数？因为这个数组是排好序的，而且我们是从最小的数字开始逐一尝试的。当`first`与`last`的总和小于`sum`时，可以确定，就算继续尝试更小的数（像`last`那样往回移动）也找不到补数。

9

为什么这么做可以找出last的所有补数？因为所有数值对必定由first和last组成。找出first的所有补数，就等于找出了last的所有补数。

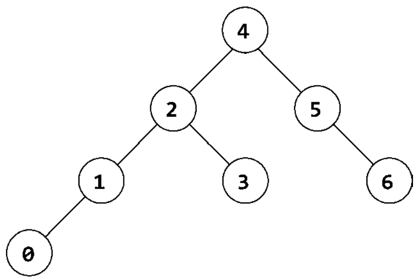
```
void printPairSums(int[] array, int sum) {
   Arrays.sort(array);
   int first = 0;
   int last = array.length - 1;
   while (first < last) {
     int s = array[first] + array[last];
     if (s == sum) {
         System.out.println(array[first] + " " + array[last]);
         first++;
        last--;
     } else {
        if (s < sum) first++;
        else last--;
     }
   }
}
```

17.13　有个简单的类似结点的数据结构BiNode，包含两个指向其他结点的指针。数据结构BiNode可用来表示二叉树（其中node1 为左子结点，node2 为右子结点）或双向链表（其中node1 为前趋结点，node2 为后继结点）。编写一个方法，将二叉查找树（用BiNode实现）转换为双向链表。要求所有数值的排序不变，转换操作不得引入其他数据结构（即直接操作原先的数据结构）。（第 105 页）

解法

此题看似复杂，不过，运用递归就能实现得相当优美。要解决此题，你需要对递归有非常深刻的理解。

设想有一棵简单的二叉查找树：



convert方法应该将它转换成下面的双向链表：

0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6

下面我们会从根结点开始（结点4），以递归方式解决问题。

我们知道，树的左右两半会在双向链表里形成它们自己的子部分（也即，它们在链表里的位置是连续的）。那么，若能以递归方式将左子树和右子树转换成双向链表，我们有办法从这些子

部分构建出最终的链表吗？

当然有！直接合并这两个子部分即可。

相关伪码大致如下：

```
BiNode convert(BiNode node) {
   BiNode left = convert(node.left);
   BiNode right = convert(node.right);
   mergeLists(left, node, right);
   return left; // 左边的开头
}
```

为了实现上述伪码的琐碎细节，我们需要取得每个链表的表头和表尾，有以下几种做法。

解法 1：附加数据结构

第一种，也是比较简单的一种方法，是创建一个NodePair数据结构，只包含链表的表头和表尾。然后，convert方法就可以返回一个NodePair对象。

下面是这种做法的实现代码。

```
private class NodePair {
   BiNode head;
   BiNode tail;

   public NodePair(BiNode head, BiNode tail)  {
     this.head = head;
     this.tail = tail;
   }
}

public NodePair convert(BiNode root) {
   if (root == null) {
     return null;
   }

   NodePair part1 = convert(root.node1);
   NodePair part2 = convert(root.node2);

   if (part1 != null) {
     concat(part1.tail, root);
   }

   if (part2 != null) {
     concat(root, part2.head);
   }

   return new NodePair(part1 == null ? root : part1.head,
                       part2 == null ? root : part2.tail);
}

public static void concat(BiNode x, BiNode y) {
   x.node2 = y;
   y.node1 = x;
}
```

上面的代码仍是在BiNode数据结构里进行转换操作，我们只是借用NodePair来返回额外的数据。另一种方式是使用有两个BiNode的数组，可实现相同的目的，但这么做会显得有些凌乱（没错，面试官喜欢干净的代码，特别是在面试中）。

做得不错，不过，要是不必用到额外的数据结构，岂不更好？是的，我们可以。

解法 2：取回表尾

之前用NodePair返回链表的表头和表尾，现在改为只返回表头，然后借助表头找到链表的表尾。

```
public static BiNode convert(BiNode root) {
  if (root == null) {
    return null;
  }

  BiNode part1 = convert(root.node1);
  BiNode part2 = convert(root.node2);

  if (part1 != null) {
    concat(getTail(part1), root);
  }

  if (part2 != null) {
    concat(root, part2);
  }

  return part1 == null ? root : part1;
}

public static BiNode getTail(BiNode node) {
  if (node == null) return null;
  while (node.node2 != null) {
    node = node.node2;
  }
  return node;
}
```

除了调用getTail，这段代码与解法1几乎完全相同，但这么做效率并不是很高。深度为d的叶结点会被getTail方法访问d次（在该叶结点之上有几个结点就访问几次），导致整体运行时间为$O(N^2)$，其中N为树的结点数。

解法 3：构造一个环状链表

在解法2的基础上，可以构建第三种也是最后一种解法。

这种做法需要用BiNode返回链表的表头和表尾。具体是将每个链表当作一个环形链表的表头返回，然后，直接调用head.node1就能取得表尾。

```
public static BiNode convertToCircular(BiNode root) {
  if (root == null) {
    return null;
  }
```

```
  BiNode part1 = convertToCircular(root.node1);
  BiNode part3 = convertToCircular(root.node2);

  if (part1 == null && part3 == null) {
    root.node1 = root;
    root.node2 = root;
    return root;
  }
  BiNode tail3 = (part3 == null) ? null : part3.node1;

  /* 将左边加至根 */
  if (part1 == null) {
    concat(part3.node1, root);
  } else {
    concat(part1.node1, root);
  }

  /* 将右边加至根 */
  if (part3 == null) {
    concat(root, part1);
  } else {
    concat(root, part3);
  }

  /* 将右边加至左边 */
  if (part1 != null && part3 != null) {
    concat(tail3, part1);
  }

  return part1 == null ? root : part1;
}

/* 将链表转换为环形链表，然后断开
 * 环形连接 */
public static BiNode convert(BiNode root) {
  BiNode head = convertToCircular(root);
  head.node1.node2 = null;
  head.node1 = null;
  return head;
}
```

注意，我们已将代码主体部分移至`convertToCircular`，`convert`方法会调用这个方法取得环形链表的表头，然后断开环状连接。

这种做法需要用时$O(N)$，因为每个结点平均只会访问一次（或者，更准确地说，是$O(1)$次）。

17.14　哦，不！你刚刚写好一篇长文，却倒霉地误用了“查找/替换”，不慎删除了文档中所有空格、标点，大写变成小写。比如，句子“I reset the computer. It still didn’t boot!”（我重启了电脑，但还没启动好！）变成了“iresetthecomputeritstilldidntboot”。你发现，只要能正确分离各个单词，加标点和调整大小写都不成问题。大部分单词在字典里都找得到，有些字符串如专有名词则找不到。

给定一个字典（一组单词），设计一个算法，找出拆分一连串单词的最佳方式。这里“最佳”的定义是，解析后无法辨识的字符序列越少越好。

举个例子，字符串“jesslookedjustliketimherbrother”的最佳解析结果为“JESS looked just like TIM her brother”，总共有 7 个字符无法辨别，全部显示为大写，以示区别。（第 105 页）

解法

有些面试官喜欢开门见山，给你具体的问题，也有的面试官喜欢告诉你一堆不必要的上下文，就像此题一样。遇到这种情况，最好将问题好好梳理一下，找出到底要做什么。

此题的关键是要找到一种方法，将字符串拆分为几个单词，使得解析后剩下的字符越少越好。

注意，我们并不打算试图去“理解”字符串，“thisisawesome”可以解析为“this is awesome”，同样可以解析为“this is a we some”。

此题的重点在于找到一种方法，从子问题的角度来定义问题解法（也即解析后的字符串）。一种做法是递归访问整个字符串。在每个时间点上，最佳解析是从两种可能决定中择优而取。

(1) 在这个字符后插入一个空格。

(2) 不在这个字符后插入一个空格。

我们将以字符串thit为例过一遍此题的解法，如下所示。为了清楚起见，我们将使用以下记号：

- 无效单词（字典里找不到的）全部**大写**；
- 有效的单词加**下划线**；
- 结合在一起的字符（字符之间没有空格）**加粗**表示。

这些在字符串里的**加粗**字符仍处于“待解析”的状态，我们还未决定这些字符是有效的还是无效的（在字典中找不找得到）。

```
p(thit)
   = min(T + p(hit), p(thit)) --> 1 inv.
     T + p(hit) = min(T + H + p(it), T + p(hit)) --> 1 inv.
        T + H + p(it) = min(T + H + i + p(t), T + H + p(it)) --> 2
           T + H + i + p(t) = T + H + i + T = 3 invalid
           T + H + p(it) = T + H + it = 2 invalid
        T + p(hit) = min(T + hi + p(t), T + p(hit)) --> 1 inv.
           T + hi + p(t) = T + hi + T = 2 invalid
           T + p(hit) = T + hit = 1 invalid
     p(thit) = min(TH + p(it), p(thit)) --> 2 inv.
        TH + p(it) = min(TH + i + p(t), TH + p(it)) --> 2 inv.
           TH + i + p(t) = TH + i + T = 3 invalid
           TH + p(it) = TH + it = 2 invalid
        p(thit) = min(THI + p(t), p(thit)) --> 4 inv.
           THI + p(t) = THI + T = 4 invalid
           p(thit) = THIT = 4 invalid
```

在上述步骤中，请注意每一层都分为两部分。第一部分分割字符串，而第二部分则进行结合。

例如，当首次调用p(thit)时，当前被解析的字符就是第一个t，会递归到两个方向。第一

个（第3行）会在t后面插入一个空白，然后试着找出解析hit的最佳方式。第二个（第10行）会试着找出t和h之间没有空白的最佳解析方式。重复执行上述动作，最终就会得到字符串所有可能的解析方式。

下面是该解法的实现代码。为了简单起见，我们实现该算法时只返回无效字符的个数。

```
public int parseSimple(int wordStart, int wordEnd) {
   if (wordEnd >= sentence.length()) {
     return wordEnd - wordStart;
   }

   String word = sentence.substring(wordStart, wordEnd + 1);

   /* 切断当前的单词 */
   int bestExact = parseSimple(wordEnd + 1, wordEnd + 1);
   if (!dictionary.contains(word)) {
     bestExact += word.length();
   }

   /* 扩展当前的单词 */
   int bestExtend = parseSimple(wordStart, wordEnd + 1);

   /* 找出最佳单词 */
   return Math.min(bestExact, bestExtend);
}
```

这段代码还可以进行两处大的优化。

- 有些递归重复了。例如，在前面的解析示例中，我们重复计算了it的最佳解析方式。其实第一次算出来后就可以将结果缓存起来，以供之后使用。运用动态规划法就可以实现这一点。
- 在某些情况下，我们或许可以预测出某一个解析将产生无效字符串。例如，假设正试着解析字符串xten，但并不存在以xt开头的单词。然而，目前的解法还是会尝试解析字符串为xt + p(en)、xte + p(n)和xten。每一次都会发现这样的单词在字典里并不存在。相反，我们应该在x后面加上空白，并从这里开始进行最佳解析。不过，怎样才能快速判断不存在以xt开头的单词呢？答案是使用trie。

下面是上述两处优化的实现代码。

```
public int parseOptimized(int wordStart, int wordEnd,
                          Hashtable<Integer, Integer> cache) {
   if (wordEnd >= sentence.length()) {
     return wordEnd - wordStart;
   }
   if (cache.containsKey(wordStart)) {
     return cache.get(wordStart);
   }

   String currentWord = sentence.substring(wordStart, wordEnd + 1);

   /* 检查前缀是否在字典里（false --> 部分匹配） */
```

9

```
    boolean validPartial = dictionary.contains(currentWord, false);

    /* 切断当前的单词 */
    int bestExact = parseOptimized(wordEnd + 1, wordEnd + 1, cache);

    /* 若完整字符串不在字典里，算作无效单词 */
    if (!validPartial || !dictionary.contains(currentWord, true)) {
      bestExact += currentWord.length();
    }

    /* 扩展当前的单词 */
    int bestExtend = Integer.MAX_VALUE;
    if (validPartial) {
      bestExtend = parseOptimized(wordStart, wordEnd + 1, cache);
    }

    /* 找出最佳单词 */
    int min = Math.min(bestExact, bestExtend);
    cache.put(wordStart, min); // 缓存结果
    return min;
}
```

注意，我们使用了散列表来缓存结果，键为单词开头的索引。也就是说，我们缓存的是字符串剩余部分的最佳解析方式。

我们可以调整代码，返回解析后的完整字符串，但这样做稍微有点复杂。我们需要使用名为`Result`的包裹类，这样才能同时返回无效字符的个数和最佳字符串。要是以C++实现的话，只需按引用传值。

```
public class Result {
    public int invalid = Integer.MAX_VALUE;
    public String parsed = “”;
    public Result(int inv, String p) {
      invalid = inv;
      parsed = p;
    }

    public Result clone() {
      return new Result(this.invalid, this.parsed);
    }

    public static Result min(Result r1, Result r2) {
      if (r1 == null) {
         return r2;
      } else if (r2 == null) {
         return r1;
      }
      return r2.invalid < r1.invalid ? r2 : r1;
    }
}

public Result parse(int wordStart, int wordEnd,
                    Hashtable<Integer, Result> cache) {
```

```
25     if (wordEnd >= sentence.length()) {
26       return new Result(wordEnd - wordStart,
27          sentence.substring(wordStart).toUpperCase());
28     }
29     if (cache.containsKey(wordStart)) {
30       return cache.get(wordStart).clone();
31     }
32     String currentWord = sentence.substring(wordStart, wordEnd + 1);
33     boolean validPartial = dictionary.contains(currentWord, false);
34     boolean validExact = validPartial &&
35                          dictionary.contains(currentWord, true);
36
37     /* 切断当前的单词 */
38     Result bestExact = parse(wordEnd + 1, wordEnd + 1, cache);
39     if (validExact) {
40       bestExact.parsed = currentWord + “ ” + bestExact.parsed;
41     } else {
42       bestExact.invalid += currentWord.length();
43       bestExact.parsed = currentWord.toUpperCase() + “ ” +
44                          bestExact.parsed;
45     }
46
47     /* 扩展当前的单词 */
48     Result bestExtend = null;
49     if (validPartial) {
50       bestExtend = parse(wordStart, wordEnd + 1, cache);
51     }
52
53     /* 找出最佳单词 */
54     Result best = Result.min(bestExact, bestExtend);
55     cache.put(wordStart, best.clone());
56     return best;
57 }
```

在动态规划问题中，对于如何缓存对象，要非常小心。若缓存的东西是对象而非基本数据类型，很可能需要复制该对象。这可以从上面第30～55行代码看出。如果不加复制，后续对parse的调用就会篡改缓存里的值。

9.18 高难度题

18.1 编写一个函数，将两个数字相加。不得使用+或其他算术运算符。（第 105 页）

解法

遇到这类问题，第一反应是我们需要跟比特位打交道，八九不离十。何出此言？原因很简单，连加号（+）都不能用了，还有其他选择吗？再说了，计算机在计算时就是跟比特位打交道的。

接下来，我们应该着眼于切实理解加法是怎么工作的。我们可以过一遍加法问题，看看自己能否悟出新东西——某种模式，然后，试试能否用代码来实现。

闲话少说，下面就来探讨一个加法问题，并以十进制运算，这样更容易理解。

要做`759 + 674`加法运算，通常会将每个数字的个位数（`digit[0]`）相加、进位，然后将每个数字的十位数（`digit[1]`）相加、进位，依此类推。二进制加法也可以采取同样的做法：各位数相加，必要时进位。

有没有办法让程序简单一点呢？当然有！设想一下，把“相加”和“进位”等步骤分开，也就是说，像下面这么做。

(1) 将759和674相加，但“忘了”进位，得到323。

(2) 将759和674相加，但只进位，不会将各位数加在一起，得到1110。

(3) 将前面两步操作的结果加起来——递归执行步骤(1)和步骤(2)描述的过程：1110 + 323 = 1433。

那么，对于二进制，该怎么做？

(1) 若将两个二进制数加在一起，但忘记进位，只要a和b的i位相同（皆为0或皆为1），总和的i位就为0。这实质上就是异或操作（XOR）。

(2) 若将两个数字加在一起，但只进位，只要a和b的i–1位皆为1，总和的i位就为1。这实质上就是位与（AND）加上移位操作。

(3) 接着，递归执行步骤(1)和(2)，直至没有进位为止。

下面是该算法的实现代码。

```
public static int add(int a, int b) {
   if (b == 0) return a;
   int sum = a ^ b; // 相加但不进位
   int carry = (a & b) << 1; // 进位，但不相加
   return add(sum, carry); // 递归
}
```

要求我们实现基本算术运算，比如加法和减法，这类问题比较常见。这些问题的关键在于深入挖掘这些运算通常是怎么实现的，这样就可根据给定问题的限制重新实现相关运算。

18.2　编写一个方法，洗一副牌。要求做到完美洗牌，换言之，这副牌 52!种排列组合出现的概率相同。假设给定一个完美的随机数发生器。（第 106 页）

解法

这个面试题非常有名，算法也很知名。掌握这个算法的人却不多，如果你还不是其中之一，还请继续往下看。

假定有个数组，含n个元素，类似如下：

[1] [2] [3] [4] [5]

利用简单构造法，我们不妨先问问自己：假定有个方法`shuffle(...)`对n − 1个元素有效，我们可以用它来打乱n个元素的次序吗？

当然可以，而且非常容易实现。我们会先打乱前n – 1个元素的次序，然后，取出第n个元素，将它与数组中的元素随机交换。就这么简单！

递归解法的算法类似如下：

```
/* lower和higher（含）之间的随机数 */
int rand(int lower, int higher) {
  return lower + (int)(Math.random() * (higher - lower + 1));
}

int[] shuffleArrayRecursively(int[] cards, int i) {
  if (i == 0) return cards;

  shuffleArrayRecursively(cards, i - 1); // 打乱先前部分的次数
  int k = rand(0, i); // 随机挑选索引进行交换

  /* 交换元素k和i */
  int temp = cards[k];
  cards[k] = cards[i];
  cards[i] = temp;

  /* 返回元素次序被打乱的数组 */
  return cards;
}
```

以迭代方式实现的话，这个算法又会是什么样？让我们先考虑一下。我们要做的就是遍历整个数组，对每个元素i，将`array[i]`与0和i（含）之间的随机元素交换。

其实，这个算法一点也不绕，很适合以迭代方式实现：

```
void shuffleArrayInteratively(int[] cards) {
  for (int i = 0; i < cards.length; i++) {
    int k = rand(0, i);
    int temp = cards[k];
    cards[k] = cards[i];
    cards[i] = temp;
  }
}
```

这样就可以用迭代法实现该算法了。

18.3　编写一个方法，从大小为n的数组中随机选出m个整数。要求每个元素被选中的概率相同。（第 106 页）

解法

与18.2类似，我们可以利用简单构造法，以递归方式处理此题。

假定有个算法可以从包含$n-1$个元素的数组中随机抽出m个元素，我们可以使用该算法从包含n个元素的数组中随机抽出m个元素吗？

我们可以先从前$n-1$个元素中随机抽出m个元素。然后，只需决定`array[n]`是否应该插入`subset`（从中随机抽出一个元素）。一种简单的做法是从0到n中随机挑选一个数k。若$k < m$，则将`array[n]`插入`subset[k]`。将`array[n]`插入`subset`（按比例概率）以及从`subset`中随机移除一个元素，两者都很“公平”。

这个递归算法的伪码大致如下：

```
int[] pickMRecursively(int[] original, int m, int i) {
```

9

```
  if (i + 1 == m) { // 终止条件
    /* 返回original数组的前m个元素 */
  } else if (i + m > m) {
    int[] subset = pickMRecursively(original, m, i - 1);
    int k = random value between 0 and i, inclusive
      if (k < m) {
        subset[k] = original[i];
      }
    return subset;
  }
  return null;
}
```

这个算法的迭代实现写起来更明晰。在这种做法中，我们会先创建数组subset，并将它初始化为original数组的前m个元素。然后，从元素m开始，迭代访问original数组，只要$k<m$，就将array[i]插入subset数组的（随机选出的）位置k。

```
int[] pickMIteratively(int[] original, int m) {
  int[] subset = new int[m];

  /* 用original数组的前m个元素填入subset */
  for (int i = 0; i < m ; i++) {
    subset[i] = original[i];
  }

  /* 访问original数组的剩余元素 */
  for (int i = m; i < original.length; i++) {
    int k = rand(0, i); // 取得0到i（含）之间的随机数
    if (k < m) {
      subset[k] = original[i];
    }
  }

  return subset;
}
```

一点也不奇怪，这两个解法与打乱数组的算法非常相似。

18.4　编写一个方法，数出 0 到n（含）中数字 2 出现了几次。（第 106 页）

解法

面对此题，我们想到的第一种做法会是，也应该是蛮力法。记住，面试官希望看到你是怎么解题的，先给出蛮力解法也是非常不错的开始。

```
/* 数一数0到n中数字2出现的次数 */
int numberOf2sInRange(int n) {
   int count = 0;
   for (int i = 2; i <= n; i++) { // 不妨直接从2开始
     count += numberOf2s(i);
   }
   return count;
}
```

```

/* 数出某个数字中有几个2 */
int numberOf2s(int n) {
   int count = 0;
   while (n > 0) {
     if (n % 10 == 2) {
        count++;
     }
     n = n / 10;
   }
   return count;
}
```

其中有个地方应该注意，就是最好将numberOf2s独立写成一个方法，这样一来，代码也许更加清晰，也会展现出你注重代码的干净齐整。

改进后的解法

之前的解法是从一个范围内的数字来看，现在我们从数字的每个位来观察问题。假设有下面一个数字序列：

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
...
110 111 112 113 114 115 116 117 118 119
```

由观察可知，每10个数字中，最后一位为2的情况大概会出现一次，因为2在连续10个数中都会出现一次。实际上，任意位为2的概率大概是1/10。

之所以说“大概”，是因为存在边界条件（非常常见）。例如，在1到100之间，十位数为2的概率正好为1/10。然而，在1到37之间，十位数为2的概率就会比1/10还大。

下面逐一分析`digit < 2`、`digit = 2`和`digit > 2`三种情况，就能算出准确的比率。

- 情况1：digit < 2

以`x = 61 523`和`d = 3`为例，可以看出`x[d] = 1`（也即x的第d位数为1）。第3位数为2的范围是`2000 - 2999`、`12 000 - 12 999`、`22 000 - 22 999`、`32 000 - 32 999`、`42 000 - 42 999`和`52 000 - 52 999`，还没到范围`62 000 - 62 999`，因此第3位数总共有6000个2。这个数量等于范围1到60 000里第3位数为2的数量。

换句话说，我们可以将原来的数往下降至最近的10^{d+1}，然后再除以10，就可以算出第d位数为2的数量。

if x[d] < 2: count2sInRangeAtDigit(x, d) =
    let y = round down to nearest 10^{d+1}
    return y / 10

- 情况2：digit > 2

现在，我们再来看看x的第d位数大于2（`x[d] > 2`）的情况。基本上，我们可以运用之前相同的逻辑，确认范围`0 - 63 525`里第3位数为2的数量与范围`0 - 70 000`是相同的。因此，之前

是往下降，现在是往上升。

if x[d] > 2: count2sInRangeAtDigit(x, d) =
  let y = round up to nearest 10^{d+1}
  return y / 10

- 情况3：digit = 2

最后这种情况可能是最棘手的，不过仍可套用之前的逻辑。以`x = 62 523`和`d = 3`为例，由之前的逻辑可得到相同的范围（也即范围`2000 - 2999`, `12 000 - 12 999`, ..., `52 000 - 52 999`）。在最后余下的`62 000 - 62 523`这个局部范围里，第3位数为2的数量有多少？其实，再显而易见不过了。只有524个（`62 000`, `62 001`, ..., `62 523`）。

if x[d] > 2: count2sInRangeAtDigit(x, d) =
  let y = round down to nearest 10^{d+1}
  let z = right side of x (i.e., x % 10^{d})
  return y / 10 + z + 1

现在，只需迭代访问数字中的每个位数。相关代码实现起来相当直接。

```
public static int count2sInRangeAtDigit(int number, int d) {
   int powerOf10 = (int) Math.pow(10, d);
   int nextPowerOf10 = powerOf10 * 10;
   int right = number % powerOf10;

   int roundDown = number - number % nextPowerOf10;
   int roundUp = roundDown + nextPowerOf10;

   int digit = (number / powerOf10) % 10;
   if (digit < 2) { // 若第digit位数……
     return roundDown / 10;
   } else if (digit == 2) {
     return roundDown / 10 + right + 1;
   } else {
     return roundUp / 10;
   }
}

public static int count2sInRange(int number) {
   int count = 0;
   int len = String.valueOf(number).length();
   for (int digit = 0; digit < len; digit++) {
     count += count2sInRangeAtDigit(number, digit);
   }
   return count;
}
```

解决此题时，需要进行非常仔细的测试，务必列全一系列的测试用例，然后逐一测试验证。

18.5　有个内含单词的超大文本文件，给定任意两个单词，找出在这个文件中这两个单词的最短距离（也即相隔几个单词）。有办法在$O(1)$时间里完成搜索操作吗？解法的空间复杂度如何？（第 106 页）

解法

在此题中，我们假设单词word1和word2谁在前谁在后无关紧要，当然最好与面试官确认能否做此假设。若考虑单词前后顺序的话，那么，下面给出的代码需要稍作调整。

要解决此题，我们只需遍历一次这个文件。在遍历期间，我们会记下最后看见word1和word2的地方，并把它们的位置存入lastPosWord1和lastPosWord2中。碰到word1时，就拿它跟lastPosWord2比较，如有必要则更新min，然后更新lastPosWord1。而每当碰到word2时，我们也会执行同样的操作。遍历结束后，就可得到最短距离。

下面是该算法的实现代码。

```java
public int shortest(String[] words, String word1, String word2) {
   int min = Integer.MAX_VALUE;
   int lastPosWord1 = -1;
   int lastPosWord2 = -1;
   for (int i = 0; i < words.length; i++) {
     String currentWord = words[i];
     if (currentWord.equals(word1)) {
        lastPosWord1 = i;
        // 若要区别单词的前后顺序，注掉下面3行
        int distance = lastPosWord1 - lastPosWord2;
        if (lastPosWord2 >= 0 && min > distance) {
          min = distance;
        }
     } else if (currentWord.equals(word2)) {
        lastPosWord2 = i;
        int distance = lastPosWord2 - lastPosWord1;
        if (lastPosWord1 >= 0 && min > distance) {
          min = distance;
        }
     }
   }
   return min;
}
```

如果上述代码要被重复调用（查询其他单词对的最短距离），可以构造一个散列表，记录每个单词及其出现的位置。然后，我们只需找出listA和listB中（算术）差值最小的那两个值。

计算listA和listB中元素最小差值有好几种方法，以下面的列表为例：

```
listA: {1, 2, 9, 15, 25}
listB: {4, 10, 19}
```

将这两个列表合并为一个列表并排序，在每个数字后面打上标记，标明取自哪个列表。打标记时可将每个值封装在一个类里，这个类有两个成员变量：data（储存实际值）和listNumber。

```
list: {1a, 2a, 4b, 9a, 10b, 15a, 19b, 25a}
```

现在，想要找出最短距离的话，只需遍历合并后的列表，查找两个取自不同列表的连续数字且它们之间的差为最小值。在上面的例子中，答案是最短距离为1（在9*a*和10*b*之间）。

18.6　设计一个算法，给定 10 亿个数字，找出最小的 100 万个数字。假定计算机内存足以容纳全部 10 亿个数字。（第 106 页）

解法

此题有很多种解法，下面将介绍其中三种：排序、小顶堆和选择排序（selection rank）。

解法 1：排序

按升序排序所有元素，然后取出前100万个数。时间复杂度为$O(n \log(n))$。

解法 2：小顶堆

我们可以使用小顶堆来解题。首先，为前100万个数字创建一个大顶堆（最大元素位于堆顶）。然后，遍历整个数列，将每个元素插入大顶堆，并删除最大的元素。

遍历结束后，我们将得到一个堆，刚好包含最小的100万个数字。这个算法的时间复杂度为$O(n \log(m))$，其中m为待查找数值的数量。

解法 3：选择排序算法（假如你可以修改原始数组）

在计算机科学中，选择排序是个很有名的算法，可以在线性时间内找到数组中第i个最小（或最大）元素。

如果这些元素各不相同，则可在预期的$O(n)$时间内找到第i个最小的元素。该算法的基本流程如下。

(1) 在数组中随机挑选一个元素，将它用作“pivot”（基准）。以pivot为基准划分所有元素，记录pivot左边的元素个数。

(2) 如果左边刚好有i个元素，则直接返回左边最大的元素。

(3) 如果左边元素个数大于i，则继续在数组左边部分重复执行该算法。

(4) 如果左边元素个数小于i，则在数组右边部分重复执行该算法，但只查找排i – leftSize的那个元素。

下面是该算法的实现代码。

```
public int partition(int[] array, int left, int right, int pivot) {
  while (true) {
    while (left <= right && array[left] <= pivot) {
       left++;
    }

    while (left <= right && array[right] > pivot) {
       right--;
    }

    if (left > right) {
       return left - 1;
```

```
    }
    swap(array, left, right);
  }
}

public int rank(int[] array, int left, int right, int rank) {
  int pivot = array[randomIntInRange(left, right)];

  /* 分割，返回左边部分的结尾 */
  int leftEnd = partition(array, left, right, pivot);

  int leftSize = leftEnd - left + 1;
  if (leftSize == rank + 1) {
    return max(array, left, leftEnd);
  } else if (rank < leftSize) {
    return rank(array, left, leftEnd, rank);
  } else {
    return rank(array, leftEnd + 1, right, rank - leftSize);
  }
}
```

一旦找到第i小的元素，你就可以遍历整个数组，找到所有小于或等于该元素的值。

如果这些元素有重复值（一般不大可能），就需要对这个算法略作调整，以适应这一变化。不过，这样一来，就不能保证算法执行时间的上限了。

有个算法可以保证在线性时间内找到第i小的元素，无论元素有无重复值。然而，这个算法的复杂度远远超出了面试的范围。若有兴趣的话，请参考CLRS四人合著的《算法导论》一书。

18.7 给定一组单词，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。（第 106 页）

解法

此题看似比较复杂，让我们先来简化一番。如果只是想知道由列表中的其他两个单词组成的最长单词，该怎么处理？

我们可以通过遍历整个列表，从最长单词到最短单词，将每个单词分割成所有可能的两半，然后检查左右两半是否在列表中。

上述做法的伪码大致如下：

```
String getLongestWord(String[] list) {
  String[] array = list.SortByLength();
  /* 创建map以便查找 */
  HashMap<String, Boolean> map = new HashMap<String, Boolean>;

  for (String str : array) {
    map.put(str, true);
  }

  for (String s : array) {
    // 切分成所有可能的两半
    for (int i = 1; i < s.length(); i++) {
```

```
          String left = s.substring(0, i);
          String right = s.substring(i);
          // 检查左右两半是否在数组中
          if (map[left] == true && map[right] == true) {
             return s;
          }
       }
    }
    return str;
}
```

若知道最长单词由另外两个单词组合而成时，这么做非常有效。但是，若单词可以由任意数量的其他单词组成，又会怎么样呢？

在这种情况下，我们可以采用非常相似的做法，只修改一处：之前会检查右半部分是否在数组中，现在改为递归检查右半部分可否由数组其他元素构建出来。

下面是该算法的实现代码：

```
String printLongestWord(String arr[]) {
   HashMap<String, Boolean> map = new HashMap<String, Boolean>();
   for (String str : arr) {
     map.put(str, true);
   }
   Arrays.sort(arr, new LengthComparator()); // 按长度排序
   for (String s : arr) {
     if (canBuildWord(s, true, map)) {
        System.out.println(s);
        return s;
     }
   }
   return “”;
}

boolean canBuildWord(String str, boolean isOriginalWord,
                     HashMap<String, Boolean> map) {
   if (map.containsKey(str) && !isOriginalWord) {
     return map.get(str);
   }
   for (int i = 1; i < str.length(); i++) {
     String left = str.substring(0, i);
     String right = str.substring(i);
     if (map.containsKey(left) && map.get(left) == true &&
         canBuildWord(right, false, map)) {
        return true;
     }
   }
   map.put(str, false);
   return false;
}
```

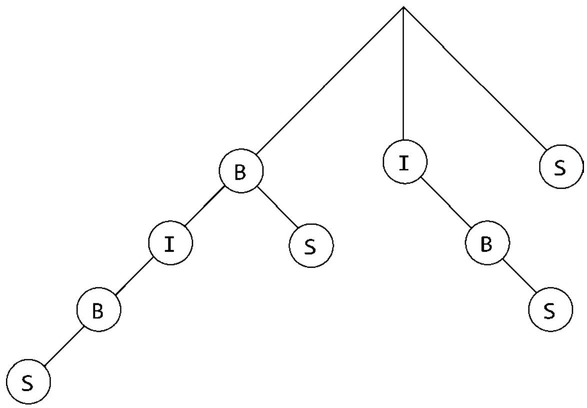
注意，在这个解法中，我们做了一个小小的优化。我们使用动态规划方法缓存了多次调用之间的结果。这样一来，如需反复检查有无办法构造“testingtester”，就只需要计算一次。

其中，布尔标志`isOriginalWord`用于完成上面的优化。调用方法`canBuildWord`时，会传入原始单词和每个子串，在算法里，第一步会先检查缓存里有无之前计算好的结果。但是，这里也有个问题：对于原始单词，`map`会将这些单词初始化为`true`，但我们又不想返回`true`（因为单词不能只由它本身组成）。因此，对于原始单词，我们会利用`isOriginalWord`标志直接跳过这项检查。

18.8 给定一个字符串s和一个包含较短字符串的数组T，设计一个方法，根据T中的每一个较短字符串，对s进行搜索。**（第 106 页）**

解法

首先，创建s的后缀树（suffix tree）。举个例子，若单词为`bibs`，则这棵树如下所示：



然后，只需在这棵后缀树中搜索查找T中的每个字符串。注意，如果“B”是个单词的话，你会得到两个位置。

```
public class SuffixTree {
   SuffixTreeNode root = new SuffixTreeNode();
   public SuffixTree(String s) {
     for (int i = 0; i < s.length(); i++) {
        String suffix = s.substring(i);
        root.insertString(suffix, i);
     }
   }

   public ArrayList<Integer> search(String s) {
     return root.search(s);
   }
}

public class SuffixTreeNode {
   HashMap<Character, SuffixTreeNode> children = new
                   HashMap<Character, SuffixTreeNode>();
   char value;
   ArrayList<Integer> indexes = new ArrayList<Integer>();
```

```
  public SuffixTreeNode() { }

  public void insertString(String s, int index) {
    indexes.add(index);
    if (s != null && s.length() > 0) {
       value = s.charAt(0);
       SuffixTreeNode child = null;
       if (children.containsKey(value)) {
          child = children.get(value);
       } else {
          child = new SuffixTreeNode();
          children.put(value, child);
       }
       String remainder = s.substring(1);
       child.insertString(remainder, index);
    }
  }

  public ArrayList<Integer> search(String s) {
    if (s == null || s.length() == 0) {
       return indexes;
    } else {
       char first = s.charAt(0);
       if (children.containsKey(first)) {
          String remainder = s.substring(1);
          return children.get(first).search(remainder);
       }
    }
    return null;
  }
}
```

18.9　随机生成一些数字并传入某个方法。编写一个程序，每当收到新数字时，找出并记录中位数。（第 106 页）

解法

一种解法是使用两个优先级堆（priority heap）：一个大顶堆，存放小于中位数的值，以及一个小顶堆，存放大于中位数的值。这会将所有元素大致分为两半，中间的两个元素位于两个堆的堆顶。这样一来，要找出中位数就是小事一桩。

不过，“大致分为两半”又是什么意思呢？“大致”的意思是，如果有奇数个值，其中一个堆就会多一个值。经观察可知，以下两点为真。

- 如果`maxHeap.size() > minHeap.size()`，则`maxHeap.top()`为中位数。
- 如果`maxHeap.size() == minHeap.size()`，则`maxHeap.top()`和`minHeap.top()`的平均值为中位数。

当要重新平衡这两个堆时，我们会确保maxHeap一定会多一个元素。

这个算法说明如下。有新的值生成时，如果这个值小于等于中位数，则放入maxHeap中，否

则放入minHeap。两个堆的元素个数相等，或者maxHeap可能多一个元素。这个限制条件很容易得到保证，不满足的话，只要从一个堆搬移一个元素到另一个堆即可。通过查看maxHeap或两个堆的堆顶元素，就能以常数时间获取中位数，而更新操作的用时为$O(\log(n))$。

```
private Comparator<Integer> maxHeapComparator;
private Comparator<Integer> minHeapComparator;
private PriorityQueue<Integer> maxHeap, minHeap;

public void addNewNumber(int randomNumber) {
   /* 注意：addNewNumber会保持下面的条件：
    * maxHeap.size() >= minHeap.size() */
   if (maxHeap.size() == minHeap.size()) {
     if ((minHeap.peek() != null) &&
        randomNumber > minHeap.peek()) {
        maxHeap.offer(minHeap.poll());
        minHeap.offer(randomNumber);
     } else {
        maxHeap.offer(randomNumber);
     }
   } else {
     if(randomNumber < maxHeap.peek()) {
        minHeap.offer(maxHeap.poll());
        maxHeap.offer(randomNumber);
     }
     else {
        minHeap.offer(randomNumber);
     }
   }
}

public static double getMedian() {
   /* maxHeap至少会跟minHeap一样大，因此，若maxHeap
    * 为空，则minHeap也为空 */
   if (maxHeap.isEmpty()) {
     return 0;
   }
   if (maxHeap.size() == minHeap.size()) {
     return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
   } else {
     /* 若maxHeap与minHeap大小不同，那么，
      * maxHeap必定多一个元素，返回maxHeap
      * 的堆顶元素 */
     return maxHeap.peek();
   }
}
```

18.10 给定两个字典里的单词，长度相等。编写一个方法，将一个单词变换成另一个单词，一次只改动一个字母。在变换过程中，每一步得到的新单词都必须是字典里存在的。（第 106 页）

解法

此题看似困难，其实只要将广度优先搜索稍作修改就可以解出来。在“图”中，每个单词的

所有分支，都是在字典里相差一个字母的单词。有趣的地方在于实现，特别是，我们能一边实现一边构建这张图吗？

可以，但是有个更简单的方法。我们可以用一张“回溯地图”。在这张回溯地图中，如果B[v] = w，则表示编辑v可得到w。到达终点单词时，可以不断地使用这张回溯地图，往回找出路径。请看下面的代码：

```
LinkedList<String> transform(String startWord, String stopWord,
     Set<String> dictionary) {
   startWord = startWord.toUpperCase();
   stopWord = stopWord.toUpperCase();
   Queue<String> actionQueue = new LinkedList<String>();
   Set<String> visitedSet = new HashSet<String>();
   Map<String, String> backtrackMap =
     new TreeMap<String, String>();

   actionQueue.add(startWord);
   visitedSet.add(startWord);

   while (!actionQueue.isEmpty()) {
     String w = actionQueue.poll();
     /* 对每个变成w只需编辑一次的单词v */
     for (String v : getOneEditWords(w)) {
        if (v.equals(stopWord)) {
           // 找到我们的单词了！ 现在往回走
           LinkedList<String> list = new LinkedList<String>();
           // 将v追加至list
           list.add(v);
           while (w != null) {
              list.add(0, w);
              w = backtrackMap.get(w);
           }
           return list;
        }
        /* 若v是个字典里的单词 */
        if (dictionary.contains(v)) {
           if (!visitedSet.contains(v)) {
              actionQueue.add(v);
              visitedSet.add(v); // 标记为已访问
              backtrackMap.put(v, w);
           }
        }
     }
   }
   return null;
}

Set<String> getOneEditWords(String word) {
   Set<String> words = new TreeSet<String>();
   for (int i = 0; i < word.length(); i++) {
     char[] wordArray = word.toCharArray();
     // 将该字母改成别的字母
     for (char c = ‘A’; c <= ‘Z’; c++) {
```

```
        if (c != word.charAt(i)) {
           wordArray[i] = c;
           words.add(new String(wordArray));
        }
     }
  }
  return words;
}
```

假设n为初始单词的长度，m为字典里相同长度的单词个数。其中while循环最多会拿出m个不同的单词，故此算法的运行时间为$O(nm)$。for循环要迭代访问整个字符串，并对每个字符施以固定次数的替换操作，时间复杂度为$O(n)$。

18.11 给定一个方阵，其中每个单元（像素）非黑即白。设计一个算法，找出四条边皆为黑色像素的最大子方阵。**（第 106 页）**

解法

和许多问题一样，此题也有难易两种解法，下面将逐一讲解。

1. “简单”解法：$O(N^4)$

我们知道最大子方阵的长度可能为N，而且$N \times N$的方阵只有一个，很容易就能检查这个方阵，符合要求则返回。

如果找不到$N \times N$的方阵，可以尝试第二大的子方阵：$(N-1) \times (N-1)$。我们会迭代所有该尺寸的方阵，一旦找到符合要求的子方阵，立即返回。如果还未找到，则继续尝试$N-2$、$N-3$，等等。由于我们是从大到小逐级搜索方阵，因此第一个找到的必定是最大的方阵。

代码具体如下：

```
Subsquare findSquare(int[][] matrix) {
   for (int i = matrix.length; i >= 1; i--) {
     Subsquare square = findSquareWithSize(matrix, i);
     if (square != null) return square;
   }
   return null;
}

Subsquare findSquareWithSize(int[][] matrix, int squareSize) {
   /* 外边为N时，里头会有(N - sz + 1)个边长
    * 为sz的方阵 */
   int count = matrix.length - squareSize + 1;

   /* 迭代所有边长为squareSize的方阵 */
   for (int row = 0; row < count; row++) {
     for (int col = 0; col < count; col++) {
        if (isSquare(matrix, row, col, squareSize)) {
           return new Subsquare(row, col, squareSize);
        }
     }
   }
   return null;
```

```
}

boolean isSquare(int[][] matrix, int row, int col, int size) {
   // 检查上边界和下边界
   for (int j = 0; j < size; j++){
     if (matrix[row][col+j] == 1) {
        return false;
     }
     if (matrix[row+size-1][col+j] == 1){
        return false;
     }
   }

   // 检查左边界和右边界
   for (int i = 1; i < size - 1; i++){
     if (matrix[row+i][col] == 1){
        return false;
     }
     if (matrix[row+i][col+size-1] == 1){
        return false;
     }
   }
   return true;
}
```

2. 预处理解法：$O(N^3)$

上面的“简单”解法之所以执行速度慢，很大一部分原因在于，每次检查一个可能符合要求的方阵，都要执行$O(N)$的工作。通过预先做些处理，就可以把`isSquare`的时间复杂度降为$O(1)$，而整个算法的时间复杂度降至$O(N^3)$。

仔细分析`isSquare`的具体用处，就会发现它只需知道特定单元下方及右边的`squareSize`项是否为零。我们可以预先以直接、迭代的方式算好这些数据。

我们从右到左、自下而上迭代访问每个单元，并执行如下计算：

```
if A[r][c] is white, zeros right and zeros below are 0
else A[r][c].zerosRight = A[r][c + 1].zerosRight + 1
     A[r][c].zerosBelow = A[r + 1][c].zerosBelow + 1
```

下面这个例子给出了一个矩阵的相关值。

(0s right, 0s below)

0,0	1,3	0,0
2,2	1,2	0,0
2,1	1,1	0,0

Original Matrix

W	B	W
B	B	W
B	B	W

现在，`isSquare`方法不必再迭代$O(N)$个元素，只需检查角落的`zerosRight`和`zerosBelow`即可。下面是该算法的实现代码。注意，`findSquare`和`findSquareWithSize`基本相同，除了前者

调用了processSquare[①]以及之后操作新的数据类型。

```
public class SquareCell {
   public int zerosRight = 0;
   public int zerosBelow = 0;
   /* 声明、getter、setter */
}

Subsquare findSquare(int[][] matrix) {
   SquareCell[][] processed = processSquare(matrix);
   for (int i = matrix.length; i >= 1; i--) {
     Subsquare square = findSquareWithSize(processed, i);
     if (square != null) return square;
   }
   return null;
}

Subsquare findSquareWithSize(SquareCell[][] processed,
int squareSize) {
   /* 与第一个算法相同 */
}


boolean isSquare(SquareCell[][] matrix, int row, int col,
                 int size) {
   SquareCell topLeft = matrix[row][col];
   SquareCell topRight = matrix[row][col + size - 1];
   SquareCell bottomLeft = matrix[row + size - 1][col];
   if (topLeft.zerosRight < size) { // 检查上边界
     return false;
   }
   if (topLeft.zerosBelow < size) { // 检查左边界
     return false;
   }
   if (topRight.zerosBelow < size) { // 检查右边界
     return false;
   }
   if (bottomLeft.zerosRight < size) { // 检查下边界
     return false;
   }
   return true;
}

 SquareCell[][] processSquare(int[][] matrix) {
   SquareCell[][] processed =
     new SquareCell[matrix.length][matrix.length];

   for (int r = matrix.length - 1; r >= 0; r--) {
     for (int c = matrix.length - 1; c >= 0; c--) {
        int rightZeros = 0;
        int belowZeros = 0;
```

① 原文误为processMatrix。——译者注

```
        // 只有单元为黑色时才需处理
        if (matrix[r][c] == 0) {
           rightZeros++;
           belowZeros++;
           // 下一列在同一行上
           if (c + 1 < matrix.length) {
              SquareCell previous = processed[r][c + 1];
              rightZeros += previous.zerosRight;
           }
           if (r + 1 < matrix.length) {
              SquareCell previous = processed[r + 1][c];
              belowZeros += previous.zerosBelow;
           }
        }
        processed[r][c] = new SquareCell(rightZeros, belowZeros);
     }
   }
   return processed;
}
```

18.12　给定一个正整数和负整数组成的$N \times N$矩阵，编写代码找出元素总和最大的子矩阵。（第 106 页）

解法

此题有很多种解法，我们先从蛮力法开始，并在此基础上进行优化。

1. 蛮力法：$O(N^6)$

跟许多“求最大值”问题一样，此题也有个简单的蛮力解法。这种解法就是直接迭代所有可能的子矩阵，计算元素总和，找出最大值。

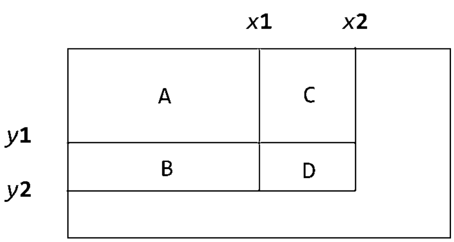
要迭代所有可能的子矩阵（且不重复），只需迭代所有的有序行配对，然后迭代所有的有序列配对。

由于要迭代$O(N^4)$个子矩阵，计算每个子矩阵的元素总和用时$O(N^2)$，因此，这个解法的时间复杂度为$O(N^6)$。

2. 动态规划法：$O(N^4)$

注意到前面的解法被拖慢了$O(N^2)$，只怪矩阵元素总和的计算太慢。有办法减少元素总和计算的用时吗？当然有！事实上，computeSum的用时可以降至$O(1)$。

考虑下面的矩形：



假设我们知道下面这些值：

```
ValD = area(point(0, 0) -> point(x2, y2))
ValC = area(point(0, 0) -> point(x2, y1))
ValB = area(point(0, 0) -> point(x1, y2))
ValA = area(point(0, 0) -> point(x1, y1))
```

每个Val*都从原点开始，在子矩形的右下角结束。

利用这些值，可得到以下等式：

```
area(D) = ValD - area(A union C) - area(A union B) + area(A)
```

或者，换一种写法：

```
area(D) = ValD - ValB - ValC + ValA
```

利用类似的逻辑，就可以有效地为矩阵里的所有点算出这些值：

```
Val(x, y) = Val(x - 1, y) + Val(x, y - 1) - Val(x - 1, y - 1) + M[x][y]
```

我们可以预先算好这些值，然后就能迅速地找到元素总和最大的子矩阵。

下面是该算法的实现代码。

```
int getMaxMatrix(int[][] original) {
   int maxArea = Integer.MIN_VALUE; // 注意，最大总和可能小于0
   int rowCount = original.length;
   int columnCount = original[0].length;
   int[][] matrix = precomputeMatrix(original);
   for (int row1 = 0; row1 < rowCount; row1++) {
     for (int row2 = row1; row2 < rowCount; row2++) {
        for (int col1 = 0; col1 < columnCount; col1++) {
           for (int col2 = col1; col2 < columnCount; col2++) {
              maxArea = Math.max(maxArea, computeSum(matrix,
                 row1, row2, col1, col2));
           }
        }
     }
   }
   return maxArea;
}

int[][] precomputeMatrix(int[][] matrix) {
   int[][] sumMatrix = new int[matrix.length][matrix[0].length];
   for (int i = 0; i < matrix.length; i++) {
     for (int j = 0; j < matrix.length; j++) {
        if (i == 0 && j == 0) { // 第一个单元
           sumMatrix[i][j] = matrix[i][j];
        } else if (j == 0) { // 第一列的单元
           sumMatrix[i][j] = sumMatrix[i - 1][j] + matrix[i][j];
        } else if (i == 0) { // 第一行的单元
           sumMatrix[i][j] = sumMatrix[i][j - 1] + matrix[i][j];
        } else {
           sumMatrix[i][j] = sumMatrix[i - 1][j] +
              sumMatrix[i][j - 1] - sumMatrix[i - 1][j - 1] +
              matrix[i][j];
        }
```

```
      }
    }
    return sumMatrix;
}

int computeSum(int[][] sumMatrix, int i1, int i2, int j1, int j2) {
    if (i1 == 0 && j1 == 0) { // 从行0、列0开始
      return sumMatrix[i2][j2];
    } else if (i1 == 0) { // 从行0开始
      return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1];
    } else if (j1 == 0) { // 从列0开始
      return sumMatrix[i2][j2] - sumMatrix[i1 - 1][j2];
    } else {
      return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1]
            - sumMatrix[i1 - 1][j2] + sumMatrix[i1 - 1][j1 - 1];
    }
}
```

3. 优化后的解法：$O(N^3)$

信不信由你，但确实有个更优的解法。如果矩阵为R行C列，我们可以在$O(R^2C)$时间内解出此题。

回想一下找出最大总和的子数组问题：给定一个整数数组，找出元素总和最大的子数组。我们有办法在$O(N)$时间内找到（元素总和）最大的子数组，该解法也可用来求解此题。

每个子矩阵都可以表示为一组连续的行和一组连续的列。如果要迭代所有连续行的组合，那么，对每一种组合找出一组可给出元素总和最大的列，就可以了。也就是说：

```
maxSum = 0
foreach rowStart in rows
    foreach rowEnd in rows
      /* 我们有一些子矩阵，rowStart为
       * 矩阵上边，rowEnd为矩阵下边，
       * 找出colStart和colEnd左右两边，
       * 使得总和最大 */
      maxSum = max(runningMaxSum, maxSum)
return maxSum
```

现在，问题转变为如何高效地找出“最好”的colStart和colEnd？此题变得越来越有意思了。

假设有如下子矩阵：

rowStart

9	-8	1	3	-2
-3	7	6	-2	4
6	-4	-4	8	-7
12	-5	3	9	-5

rowEnd

我们想要找到相应的colStart和colEnd，使得rowStart为上边、rowEnd为下边的子矩阵元素总和最大。为此，我们可以把每一列加起来，然后应用此题开头解释过的maxSubArray函数。

在前面的例子中，总和最大的子数组是第1列到第4列。这就意味着最大子矩阵为(rowStart, first column)到(rowEnd, fourth column)。

至此，可写出大致如下的伪码。

```
1  maxSum = 0
2  foreach rowStart in rows
3     foreach rowEnd in rows
4       foreach col in columns
5          partialSum[col] = sum of matrix[rowStart, col] through
6                            matrix[rowEnd, col]
7       runningMaxSum = maxSubArray(partialSum)
8     maxSum = max(runningMaxSum, maxSum)
9  return maxSum
```

第5、6行计算总和需用时R*C（要循环访问rowStart至rowEnd），因此全部用时$O(R^3C)$。不过，大功尚未告成。

在第5、6行，从头将a[0]...a[i]加起来，即使在外层for循环的前一次迭代时已计算过a[0]...a[i-1]的总和。这部分重复的计算完全可以砍掉不要。

```
maxSum = 0
foreach rowStart in rows
   clear array partialSum
   foreach rowEnd in rows
     foreach col in columns
        partialSum[col] += matrix[rowEnd, col]
     runningMaxSum = maxSubArray(partialSum)
   maxSum = max(runningMaxSum, maxSum)
return maxSum
```

最终，完整的代码大致如下：

```
public void clearArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
     array[i] = 0;
   }
}

public static int maxSubMatrix(int[][] matrix) {
   int rowCount = matrix.length;
   int colCount = matrix[0].length;

   int[] partialSum = new int[colCount];
   int maxSum = 0; // 最大总和是个空矩阵

   for (int rowStart = 0; rowStart < rowCount; rowStart++) {
     clearArray(partialSum);

     for (int rowEnd = rowStart; rowEnd < rowCount; rowEnd++) {
        for (int i = 0; i < colCount; i++) {
```

```
            partialSum[i] += matrix[rowEnd][i];
         }

         int tempMaxSum = maxSubArray(partialSum, colCount);

         /* 如欲追踪坐标，将相关代码
          * 放在这里 */
         maxSum = Math.max(maxSum, tempMaxSum);
      }
   }
   return maxSum;
}

public static int maxSubArray(int array[], int N) {
   int maxSum = 0;
   int runningSum = 0;

   for (int i = 0; i < N; i++) {
     runningSum += array[i];
     maxSum = Math.max(maxSum, runningSum);

     /* 若runningSum < 0，就没必要再继续了。
      * 重置 */
     if (runningSum < 0) {
        runningSum = 0;
     }
   }
   return maxSum;
}
```

此题非常复杂，若没有面试官的大量提示和帮助，在面试中很难周全地解出整个问题。

18.13　给定一份几百万个单词的清单，设计一个算法，创建由字母组成的最大矩形，其中每一行组成一个单词（自左向右），每一列也组成一个单词（自上而下）。不要求这些单词在清单里连续出现，但要求所有行等长，所有列等高。（第 106 页）

解法

很多与字典有关的问题，通过预先做些处理就可以解出来。对于此题，哪一部分可以做预处理呢？

好吧，如果要创建一个单词矩形，就必须满足以下要求：每一行等长，每一列等高。因此，我们可以将这个字典的单词按长短进行分组，姑且把这个分组叫作`D`，其中`D[i]`包含长度为*i*的单词串。

接下来，观察要找的最大矩形。可能形成的绝对最大的矩形有多大呢？它会是`length(largest word)2`。

```
int maxRectangle = longestWord * longestWord;
for z = maxRectangle to 1 {
   for each pair of numbers (i, j) where i*j = z {
     /* 试着用单词构建矩形，成功则返回 */
   }
}
```

从最大可能的矩形迭代至最小的矩形，可以保证第一个找到的符合要求的矩形就是题目要求的最大矩形。

现在，轮到困难的部分：`makeRectangle(int l, int h)`。这个方法试图构建长*l*高*h*的单词矩形。

一种做法是迭代所有长*h*的有序单词集合，然后检查每一列字母是否形成有效单词。这么做也行得通，但是非常低效。

假设我们正试着构造6×5的矩形，前几行单词如下：

```
there
queen
pizza
.....
```

至此可知，第一列开头几个字母为`tqp`。我们知道或者说应该知道，字典里没有以`tqp`开头的单词。既然明摆着最终创建不出有效的矩形，为何还要自寻烦恼，继续构造下去呢？

这就引出一个更优的解法。我们可以构建一棵单词查找树（trie），从而轻易查出某个子串是否为字典里单词的前缀。然后，在一行一行自上而下构造矩形时，检查每一列字母是否均为有效前缀。如果不是，则立即失败并中止，不再继续构造这个矩形。

下面是该算法的实现代码，长且复杂，下面我们会逐步解说。

一开始会做些预处理，将单词按长度分组。我们会创建一个单词查找树（每一个trie包含某长度的单词）数组，但直到真正需要时，才会构建单词查找树。

```
WordGroup[] groupList = WordGroup.createWordGroups(list);
int maxWordLength = groupList.length;
Trie trieList[] = new Trie[maxWordLength];
```

`maxRectangle`方法是代码的“主体”，从可能的最大矩形（`maxWordLength2`）开始，然后试着构建该大小的矩形。若构建失败，该方法会将最大面积减一，并尝试新的、较小的尺寸。由此，第一个成功构建的矩形必定是最大的。

```
Rectangle maxRectangle() {
   int maxSize = maxWordLength * maxWordLength;
   for (int z = maxSize; z > 0; z--) { // 从最大面积开始
     for (int i = 1; i <= maxWordLength; i++) {
        if (z % i == 0) {
           int j = z / i;
           if (j <= maxWordLength) {
             /* 构造长度i、高度j的矩形。注意，
              * i * j = z */
             Rectangle rectangle = makeRectangle(i, j);
             if (rectangle != null) {
               return rectangle;
             }
           }
        }
     }
   }
   return null;
}
```

maxRectangle又调用了makeRectangle方法，用于构造指定长度和高度的矩形。

```
Rectangle makeRectangle(int length, int height) {
   if (groupList[length-1] == null ||
       groupList[height-1] == null) {
     return null;
   }

   /* 若不存在，就构建该单词长度的trie */
   if (trieList[height - 1] == null) {
     LinkedList<String> words = groupList[height - 1].getWords();
     trieList[height - 1] = new Trie(words);
   }

   return makePartialRectangle(length, height,
                               new Rectangle(length));
}
```

makePartialRectangle方法真正负责构建矩形，参数为预期的最终长度和高度以及部分成形的矩形。如果矩形的高度已达到最后想要的高度，就直接查看每一列能否构成有效、完整的单词，然后返回。

否则，检查每一列字母能否构成有效前缀。如若不能，就立即中止，因为这个部分成形的矩形最后不可能构建出有效的矩形。

不过，如果到目前为止一切顺利，所有列都是有效的单词前缀，那么，就继续搜索相应长度的单词，追加至当前矩形的后面，然后进入递归试着以{追加中新单词的矩形}为基础构建矩形。

```
Rectangle makePartialRectangle(int l, int h, Rectangle rectangle) {
   if (rectangle.height == h) { // 检查矩形是否已完成
     if (rectangle.isComplete(l, h, groupList[h - 1])) {
        return rectangle;
     } else {
        return null;
     }
   }

   /* 将所有列与trie比较，检查是否有效 */
   if (!rectangle.isPartialOK(l, trieList[h - 1])) {
     return null;
   }

   /* 迭代访问该长度的所有单词，并加入
    * 当前的部分矩形，然后试着递归构建出
    * 矩形 */
   for (int i = 0; i < groupList[l-1].length(); i++) {
     /* 当前矩形加上新单词构建新矩形 */
     Rectangle orgPlus =
        rectangle.append(groupList[l-1].getWord(i));

     /* 试着以这个新的、部分矩形构建新矩形 */
     Rectangle rect = makePartialRectangle(l, h, orgPlus);
     if (rect != null) {
```

```
        return rect;
     }
   }
   return null;
}
```

Rectangle类代表一个部分或完整的单词矩形，可以调用方法isPartialOk来检查矩形到目前为止是否有效（即每一列都是有效的单词前缀）。方法isComplete的功能类似，不过只检查每一列是否为完整的单词。

```
public class Rectangle {
   public int height, length;
   public char [][] matrix;

   /* 构造一个“空”的矩形，长度是固定的，
    * 但高度会随着单词的加入而变化 */
   public Rectangle(int l) {
     height = 0;
     length = l;
   }

   /* 根据指定长度和高度的字符数组
    * 构造矩形，使用指定的字母矩阵
    * 表示（假定参数指定的长度和高
    * 度与数组参数的大小
    * 相符） */
   public Rectangle(int length, int height, char[][] letters) {
     this.height = letters.length;
     this.length = letters[0].length;
     matrix = letters;
   }

   public char getLetter (int i, int j) { return matrix[i][j]; }
   public String getColumn(int i) { ... }

   /* 检查所有列是否都为有效。所有列已知为
    * 有效的，因为它们是直接从字典里取出的 */
   public boolean isComplete(int l, int h, WordGroup groupList) {
     if (height == h) {
        /* 检查每一列是否为字典里的单词 */
        for (int i = 0; i < l; i++) {
           String col = getColumn(i);
           if (!groupList.containsWord(col)) {
              return false;
           }
        }
        return true;
     }
     return false;
   }

   public boolean isPartialOK(int l, Trie trie) {
     if (height == 0) return true;
```

```
    for (int i = 0; i < l; i++ ) {
       String col = getColumn(i);
       if (!trie.contains(col)) {
          return false;
       }
    }
    return true;
  }

  /* 在当前矩形上追加s来新建
   * Rectangle */
  public Rectangle append(String s) { ... }
}
```

WordGroup类是个简单的容器，包含某长度的所有单词。为方便查找，我们会将单词储存在散列表和ArrayList中。

WordGroup中的列表由静态方法createWordGroups创建。

```
public class WordGroup {
  private Hashtable<String, Boolean> lookup =
    new Hashtable<String, Boolean>();
  private ArrayList<String> group = new ArrayList<String>();

  public boolean containsWord(String s) {
    return lookup.containsKey(s);
  }

  public void addWord (String s) {
    group.add(s);
    lookup.put(s, true);
  }

  public int length() { return group.size(); }
  public String getWord(int i) { return group.get(i); }
  public ArrayList<String> getWords() { return group; }

  public static WordGroup[] createWordGroups(String[] list) {
    WordGroup[] groupList;
    int maxWordLength = 0;
    /* 找出最长单词的长度 */
    for (int i = 0; i < list.length; i++) {
       if (list[i].length() > maxWordLength) {
          maxWordLength = list[i].length();
       }
    }

    /* 将字典里的单词按长度分组，相同长度的分为一组。
     * groupList[i]会包含一串单词，每个单词的长度为
     * length (i+1) */
    groupList = new WordGroup[maxWordLength];
    for (int i = 0; i < list.length; i++) {
       /* 这里是wordLength - 1而非wordLength，
        * 因为这里是索引，不存在长度为0的单词 */
```

```
            int wordLength = list[i].length() - 1;
            if (groupList[wordLength] == null) {
               groupList[wordLength] = new WordGroup();
            }
            groupList[wordLength].addWord(list[i]);
         }
         return groupList;
      }
   }
```

此题完整代码（包括Trie和TrieNode的），可在本书所附的源码包里找出。注意，面对复杂如是的问题，你很可能只需要写出伪码即可。毕竟，要在这么短的时间内写出全部代码几乎是不可能的。

索　　引

M

N

P

S

T

W

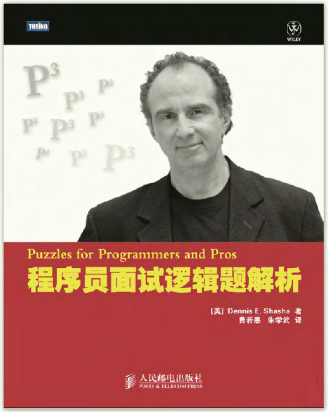
X

Y

Z

相关图书


TURING
金领简历
敲开苹果、微软、谷歌的大门
Gayle Laakmann McDowell 著
人民邮电出版社
POSTS & TELECOM PRESS


TURING
Puzzles for Programmers and Pros
程序员面试逻辑题解析
[美] Dennis E. Shasha 著
贵若愚 朱学武 译
人民邮电出版社

“如果你正打算参加技术面试，我极力推荐你阅读此书。这本书汇总了诸多你不可不知的决胜于技术面试的问题、策略和方法。”

——Ginnie，亚马逊评论者

► 150个编程题问答

从二叉树到二分查找，该部分涵盖了关于数据结构和算法的最常见、最有用的面试题以及最为精巧的解决方案。

► 应对棘手算法题的5种行之有效的方法

通过这5种方法，你可以学会如何处理并攻克算法难题，包括那些最棘手的算法题。

► 面试者最容易犯的10个错误

不要因为这些常见的错误而与成功失之交臂。要了解面试者常犯的一些错误，学会如何避免这些问题。

► 面试准备的若干策略

不要因为沉溺在无穷无尽的面试题中而错过了最重要的求职建议。这些策略和步骤可以让你更有效地准备面试。

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186 转 604
分类建议 计算机/程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33291-2

ISBN 978-7-115-33291-2
定价:59.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn